Other Things Besides Number

Abstraction, Constraint Propagation,
and String Variable Types

Joseph Scott

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in ITC room 2446, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Monday, 14 March 2016 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Michel Rueher (University of Nice Sophia Antipolis - CNRS).

Abstract

Scott, J. 2016. Other Things Besides Number. Abstraction, Constraint Propagation, and String Variable Types. *Uppsala Dissertations from the Faculty of Science and Technology* 120. 210 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9464-3.

Constraint programming (CP) is a technology in which a combinatorial problem is modeled declaratively as a conjunction of constraints, each of which captures some of the combinatorial substructure of the problem. Constraints are more than a modeling convenience: every constraint is partially implemented by an inference algorithm, called a propagator, that rules out some but not necessarily all infeasible candidate values of one or more unknowns in the scope of the constraint. Interleaving propagation with systematic search leads to a powerful and complete solution method, combining a high degree of re-usability with natural, high-level modeling.

A propagator can be characterized as a sound approximation of a constraint on an abstraction of sets of candidate values; propagators that share an abstraction are similar in the strength of the inference they perform when identifying infeasible candidate values. In this thesis, we consider abstractions of sets of candidate values that may be described by an elegant mathematical formalism, the Galois connection. We develop a theoretical framework from the correspondence between Galois connections and propagators, unifying two disparate views of the abstraction-propagation connection, namely the oft-overlooked distinction between representational and computational over-approximations. Our framework yields compact definitions of propagator strength, even in complicated cases (i.e., involving several types, or unknowns with internal structure); it also yields a method for the principled derivation of propagators from constraint definitions.

We apply this framework to the extension of an existing CP solver to constraints over strings, that is, words of finite length. We define, via a Galois connection, an over-approximation for bounded-length strings, and demonstrate two different methods for implementing this overapproximation in a CP solver. First we use the Galois connection to derive a bounded-length string representation as an aggregation of existing scalar types; propagators for this representation are obtained by manual derivation, or automated synthesis, or a combination. Then we implement a string variable type, motivating design choices with knowledge gained from the construction of the over-approximation. The resulting CP solver extension not only substantially eases modeling for combinatorial string problems, but also leads to substantial efficiency improvements over prior CP methods.

Keywords: constraint programming, string constraint problems, Galois connections, abstraction, constraint propagation, computer-aided verification applications

Joseph Scott, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Joseph Scott 2016

ISSN 1104-2516

ISBN 978-91-554-9464-3

urn:nbn:se:uu:diva-273311 (<http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-273311>)

*Dedicated to Vicki, Myrtella, Mildred, Lucille, Susie, Nancy,
Jennifer, Soren, Molly, Joëlle, Mia, Nora, and Sanna.
Because girls kick ass.*

Contents

Introduction	13
Overview	13
Contributions	23
How to Read This Thesis	25
1 Constraint Programming: An Overview	26
1.1 Constraint Problems	26
1.2 Constraint Programming Solvers	27
Part I: Abstract Consistency	33
2 Preliminaries	37
2.1 Lattice Theory and Galois Connections	37
2.1.1 Partially Ordered Sets	38
2.1.2 Lattices	39
2.1.3 Closure and Reduction Operators	40
2.1.4 Galois Connections	40
2.1.5 Sound Approximation	43
2.2 Propagator Functions	45
2.2.1 Consistency and Completeness	46
2.2.2 Exact Consistency	48
2.3 Related Work	50
2.3.1 Abstraction-based Frameworks	51
2.3.2 Closure-based Frameworks	52
2.4 A Sketch of a Galois-based Framework	52
3 Propagator Strength in the Abstract	55
3.1 $\langle \alpha, \gamma \rangle$ -Consistency	55
3.2 Propagator Completeness	58
3.2.1 Smallest Sound Approximations of Propagators	58
3.2.2 $(\gamma \circ \alpha)$ -Completeness	60
3.2.3 $(\gamma \circ \alpha)$ -Canonicity	61
3.2.4 ϕ -Canonicity and Idempotency	61
4 Cartesian Consistencies	65
4.1 Cartesian Approximation	65
4.2 Domain Consistency	66

4.3	Relaxations of Domain Consistency	69
4.3.1	Cartesian $(\gamma \circ \alpha)$ -Canonical Propagator Functions	69
4.3.2	Cartesian $\langle \alpha, \gamma \rangle$ -Consistency	70
5	Bounds Consistencies	73
5.1	Interval Approximation	73
5.2	Bounds Consistency	74
5.2.1	Bounds Consistency for Partially-Ordered Domains	75
5.2.2	Bounds Consistency for Totally-Ordered Domains	78
5.3	Other Forms of Bounds Consistency	83
5.3.1	Bounds($\mathbb{R}$) Consistency	84
5.3.2	Bounds(D) Consistency and Range Consistency	85
6	Combinations of Consistencies	89
6.1	Mixed-Type Consistency	89
6.2	Non-Uniform Consistency	91
	Part II: Constraints on String Variables	93
7	String Variables and Constraints	97
7.1	String Variables	98
7.2	String Constraints	98
7.2.1	Pure String Constraints	99
7.2.2	Mixed String Constraints	101
7.2.3	Language Membership Constraints	102
7.3	Related Work	103
7.3.1	Fixed-length String Variables	103
7.3.2	Unbounded-length String Variables	104
7.3.3	Bounded-length String Variables	107
8	Over-Approximating a String Domain	109
8.1	Bounded-Length Sequence Representation	109
8.2	Over-Approximations as Domain Representations	114
8.2.1	Consistency	114
8.2.2	Representation Invariants	114
8.3	Propagation for Bounded-Length Strings	117
8.3.1	Pure String Constraints	117
8.3.2	Mixed String Constraints	120
9	Aggregate Implementation of Bounded-length String Variables	124
9.1	Aggregate Bounded-Length Strings	124
9.1.1	Empty Sets of Candidate Characters	126
9.1.2	Representation Invariant	126
9.1.3	Aggregate Propagator Functions	127
9.2	Automatic Generation of Propagator Algorithms	129
9.2.1	Synthesis	130

9.2.2	Translation	133
10	A Bounded-length String Variable Type	137
10.1	Overview of the Variable Design	137
10.1.1	Motivating Example: Concatenation	140
10.1.2	Solver-based Requirements	144
10.1.3	Domain-based Requirements	145
10.2	Restriction Operations	146
10.2.1	on String Variables	147
10.2.2	on Bounded-length String Variables	148
10.2.3	Equality Semantics	149
10.3	Data Structure	157
10.3.1	Length	159
10.3.2	Symbols	160
10.3.3	Sequence	160
10.4	Propagation Events	162
11	Experimental Evaluation	166
11.1	Experimental Methodology	166
11.1.1	Models and Implementations	166
11.1.2	Search	171
11.2	Benchmarks	172
11.3	Experiments	174
11.3.1	Effect of Block Size	174
11.3.2	Runtime Comparisons	178
Part III: Conclusion		183
Acknowledgments		188
My Publications		190
Sammanfattning på svenska		191
Bibliography		197

List of Figures

1:	Example: Sudoku	14
2:	Constraint model of a Sudoku problem	15
3:	Solving sudoku: propagation and backtracking	17
1.1:	From constraint semantics to propagator specification	28
2.1:	Sound approximation	44
2.2:	Composition of smallest sound approximations	44
2.3:	Three views of propagation	51
3.1:	Consistency based on a Galois connection	57
3.2:	Idempotency conditions for propagator functions	62
5.1:	Implication graph of covered consistencies	84
5.2:	Bounds(D)-completeness and bounds($\mathbb{Z}$)-completeness	87
9.1:	From constraint semantics to propagator specification (repeated) ..	129
9.2:	Indexical checkers for EQ and LEN	131
9.3:	Propagator descriptions for EQ and LEN	132
9.4:	Indexical translation of the string representation invariant	132
9.5:	Improved propagator description for EQ	133
9.6:	Translated propagator descriptions for EQ and LEN	135
10.1:	Data structure for a bounded-length string variable domain	138
10.2:	Variable implementations and propagator algorithms	139
10.3:	Events for string variables	163
11.1:	Padded-string model for a CAT constraint.	170
11.2:	Effect of block size on runtime, first solution	175
11.3:	Effect of block size on runtime, branching on minimum length	176
11.4:	Effect of block size on runtime, branching on median length	177
11.5:	Runtime comparison on benchmark of NORN	179
11.6:	Runtime comparison on benchmark of NORN	180
11.7:	Exempel: Sudoku	192
11.8:	Villkorsmodell av ett sudoku	193

Introduction

[The Analytical Engine] might act upon other things besides *number*, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine.

AUGUSTA ADA KING,
COUNTESS OF LOVELACE

In 1843, Augusta Ada King, Countess of Lovelace (or, as she is more popularly styled today, Ada Lovelace), wrote a detailed exposition of the workings of a complicated, programmable, mechanical computation device: the Analytical Engine. The machine's inventor, Charles Babbage, spent many years before and after 1843 painstakingly working out the remarkable design of his Engine, which, sadly, only ever existed as an idea. Babbage thought of the Engine as a way to mechanically produce complicated mathematical tables, by executing a series of mathematical operations specified by punch cards.

Lovelace saw something else in the Engine. Spending months composing a clear explanation of the Engine, she came to see it as we do today: the first universal computer. As the epigraph that introduces this chapter indicates,¹ she realized there was a level of *abstraction* between *how* the Engine operated and *what* it did. The hardware of the Analytical Engine would have performed arithmetic operations on numbers, but Lovelace saw this as an implementation detail: the Engine could solve a non-numerical problem, as long as it could be properly expressed in terms of operations the machine could perform.

Overview

To cast this dissertation in Lovelace's terms: here the Engine is a constraint solver taking the *constraint programming* approach; the other things (besides number) are *strings*; and between them is a particular type of abstraction called a *Galois connection*.

¹Menabrea, Luigi Federico "Sketch of the Analytical Engine Invented by Charles Babbage". *Bibliothèque Universelle de Genève*, October 1842, No. 82. Translated and annotated by Augusta Ada King, Countess of Lovelace. In: Taylor, Richard. *Scientific Memoirs* 3. London: Richard and John E. Taylor, 1842, pages 666–731. Available online: www.fourmilab.ch/babbage/sketch.html.


```

1 array[1..9,1..9] of var 1..9: puzzle;
2 constraint % All different in rows:
3   forall (i in 1..9) (
4     alldifferent( [ puzzle[i,j] | j in 1..9 ] ) );
5
6 constraint % All different in columns:
7   forall (j in 1..9) (
8     alldifferent( [ puzzle[i,j] | i in 1..9 ] ) );
9
10 constraint % All different in sub-squares:
11   forall (a, o in 1..3) (
12     alldifferent( [ puzzle[(a-1)*3 + a1, (o-1)*3 + o1] |
13       a1, o1 in 1..3 ] ) );

```

Figure 2. A constraint model of the Sudoku problem, written in the MINIZINC modeling language. Adapted from [66].

- Each unknown value in the problem, namely each of the 81 cells in the grid, is called a *variable*.
- Each variable has a set of possible values, called the *domain* of the variable. In Sudoku, the initial domain of empty cells is the set $\{1, \dots, 9\}$.
- Solutions are distinguished from non-solutions by *constraints*, which are limitations to the combinations of values that variables can take.

In Sudoku, each of the three rules applies to nine groups of nine cells each, so in total the rules can be expressed as 27 constraints, each on nine variables. While the three Sudoku rules are different from each other, they are also very similar: each rule defines a group of cells that must *all* take *different* values. By generalizing the rules, we can see that the Sudoku problem is composed of 27 instances of the same *combinatorial substructure*, each for a different group of the variables. In fact, this particular bit of combinatorial substructure crops up in all sorts of combinatorial patterns: in constraint programming, it is called the ALLDIFFERENT constraint. Identifying these recurring bits of combinatorial substructure gives us a convenient way to express combinatorial problems. The ALLDIFFERENT constraint is but one example of the hundreds of useful constraints that have been identified and are described in the *Global Constraint Catalogue* [10]. A CSP can be described, in a fully declarative way, as a conjunction of these constraints, which is called a *constraint model*. A constraint model of the Sudoku problem, written in the MINIZINC modeling language [72], is shown in Figure 2. The model is a restatement of the original rules (modulo a bit of math) as ALLDIFFERENT constraints; real-world problems tend to combine constraints for *multiple* combinatorial substructures.

In CP, constraints are not just a convenient way of modeling a problem, they are also central to the way that the problem is solved. For example, while there are different ways one could try to solve a Sudoku, one method goes something like this: consider each rule, one at a time, and discard any values in cells of

the grid that violate that rule; repeat as needed. For example, we could first consider the rule “every cell in column #8 must be different.” The effect of applying this rule is shown in Figure 1(b). The possible values for each cell in column #8 (shaded) are shown in a 3×3 grid, with values for each cell that have been forbidden by the constraint ALLDIFFERENT (column #8) greyed out. There are three cells in column #8 that already have a value, so we can discard the 1, 5, and 6 from all the other cells in the column, leaving the set $\{2, 3, 4, 7, 8, 9\}$.

In constraint programming, removing values that violate a constraint from the domains of variables is called *pruning* the domains, and a procedure that prunes domains according to a particular constraint is called a *propagator* for that constraint.

Say we next select the rule for the top row of the puzzle. A propagator for the constraint for this rule would prune the domains of the variables in that row as shown in Figure 3(a), removing the values 3, 4, 6, 7, and 9 from the domain of each variable in that row. One of the resulting domains is smaller than the others: the cell in the eighth column of the top row already had some values discarded by the first propagator, so only the values in $\{2, 8\}$ remain. Each propagator only knows one of the 27 constraints in the Sudoku problem (we say that each propagator reasons about the problem *locally*) but propagators communicate with each other by removing values from variable domains. In our case, the propagator for the constraint ALLDIFFERENT (row #1) would not know that 5 is not a possible value for the cell in column #8, the same way that it cannot eliminate the value 5 from any of the other cells in row #1. But the propagator for ALLDIFFERENT (column #8) has already removed that value.

Continuing, we next consider the propagator ALLDIFFERENT for the north-east region (shaded), which, as shown in Figure 3(b), removes the values 1, 2, 3, and 4 from the cells in that region. The cell in the eighth column of the top row has one remaining value, 8, so we say that the variable for that cell is *assigned*. An assigned variable is a new clue for the Sudoku, which may allow some constraints previously checked to prune more; for example, it is now possible to eliminate the value 8 from the domains of all variables in the same row or column as the newly assigned variable.

We can continue this process for some time, considering which domains have been pruned and using that information to select another propagator. A procedure that performs this routine is called a *fixpoint algorithm*, because eventually the procedure will reach a point where none of the propagators can prune any more values. The fixpoint for our Sudoku problem is shown in Figure 3(c). There are only four cells left unassigned, but none of the 27 constraints allows us to decide between the two values that each cell could take.

To continue further, we have to make a guess about one of the values. We call this choice a *decision*: for example, we pick one value from the domain of one of the unassigned variables. Figure 3(d) shows what happens if we pick the value 4 for the circled cell. Now that one more variable is assigned, the other

9	<div>123456789</div>	6	<div>123456789</div>	7	<div>123456789</div>	4	<div>123456789</div>	3
			4			2	<div>123456789</div>	
	7			2	3		1	
5						1	<div>123456789</div>	
	4		2		8		6	
		3					<div>123456789</div>	5
	3		7				5	
		7			5		<div>123456789</div>	
4		5		1		7	<div>123456789</div>	8

(a) result of ALLDIFFERENT (Row #1)

9	<div>123456789</div>	6	<div>123456789</div>	7	<div>123456789</div>	4	<div>123456789</div>	3
			4			2	<div>123456789</div>	<div>123456789</div>
	7			2	3	<div>123456789</div>	1	<div>123456789</div>
5						1	<div>123456789</div>	
	4		2		8		6	
		3					<div>123456789</div>	5
	3		7				5	
		7			5		<div>123456789</div>	
4		5		1		7	<div>123456789</div>	8

(b) result of ALLDIFFERENT (NE region)

9	2	6	5	7	1	4	8	3
3	5	1	4	8	6	2	7	9
8	7	4	9	2	3	5	1	6
5	8	2	3	6	7	1	9	4
1	4	9	2	5	8	3	6	7
7	6	3	1	<div>123456789</div>	<div>123456789</div>	8	2	5
2	3	8	7	<div>123456789</div>	<div>123456789</div>	6	5	1
6	1	7	8	3	5	9	4	2
4	9	5	6	1	2	7	3	8

(c) fixpoint of all constraints

9	2	6	5	7	1	4	8	3
3	5	1	4	8	6	2	7	9
8	7	4	9	2	3	5	1	6
5	8	2	3	6	7	1	9	4
1	4	9	2	5	8	3	6	7
7	6	3	1	<div>123456789</div>	<div>123456789</div>	8	2	5
2	3	8	7	<div>123456789</div>	<div>123456789</div>	6	5	1
6	1	7	8	3	5	9	4	2
4	9	5	6	1	2	7	3	8

(d) take decision #1

9	2	6	5	7	1	4	8	3
3	5	1	4	8	6	2	7	9
8	7	4	9	2	3	5	1	6
5	8	2	3	6	7	1	9	4
1	4	9	2	5	8	3	6	7
7	6	3	1	<div>123456789</div>	<div>123456789</div>	8	2	5
2	3	8	7	<div>123456789</div>	<div>123456789</div>	6	5	1
6	1	7	8	3	5	9	4	2
4	9	5	6	1	2	7	3	8

(e) backtrack to (c), then take decision #2

Figure 3. Solving a sudoku by propagation and backtracking search.

propagators have enough information to assign all the remaining variables. By construction, the values assigned to the cells satisfy all the constraints, and therefore are a solution to the problem. But is this the only solution? To find out, we can *backtrack*: we return to the way the puzzle looked *before* we made our last decision (i.e., to Figure 3(c)), and then we make a *different* decision. In this case, the only other decision for the selected variable is to pick the value 9; as Figure 3(e) shows, this decision also leads to a solution, but in general a next solution might not exist.

This procedure of making decisions, propagating again, and later backtracking, is called *backtracking search*. Backtracking search is *complete*, meaning that eventually it will find every possible solution to the problem, if any solution exists. In this Sudoku problem, where we only had one decision point, each decision led directly to a solution, but in general further decision points may be necessary to find solutions. Backtracking search is complete even if there are decisions that do not lead to any solutions: once the search runs out of possible solutions, it can just backtrack, make a different decision, and continue.

Sudoku exemplifies the three elements that taken together make up a CP solver: constraints for high-level modeling of combinatorial problems, propagators for the computation of fixpoints, and backtracking search to exhaustively find all solutions to the problem. CP solvers provide fixpoint algorithms and backtracking search that are carefully designed to be both fast and correct, but also fully generic: neither the fixpoint algorithm nor the search depends on the nature of the propagators, so long as the propagators obey certain simple rules. As a result, CP solvers can be highly modular, and are easy to extend. In this dissertation, we focus on CP solvers for variables over *finite* domains.

This Sudoku example is clearly a toy problem, but this should not be taken to mean that CP applies only to puzzles. The same principles apply to real-world applications that vastly exceed this example in scope and complexity. Furthermore, while a solution to a Sudoku involves only integers, CP solvers also provide variables for other types of unknowns, such as Booleans, real numbers, and (multi)sets of integers. In the next section, we consider another variable type for CP solvers, namely string variables.

Before proceeding, the concepts of this section are summarized in the following definition:

Definition: Constraint Programming

Constraint programming (CP) is a technology for modeling and solving combinatorial problems. In CP, a combinatorial problem is fully declaratively modeled as a number of unknown *variables*, each with a set of candidate values called a *domain*, subject to *constraints* that each restrict the values of the domains for some of the variables. Constraints are reusable components that capture recurring combinatorial substructures. The conjunction of all constraints modeling a problem intensionally represents the set of all *solutions* to the problem, i.e., a set of tuples of variable values, where each tuple satisfies all the constraints. Direct computation of this solution set is generally impractical; instead each constraint is over-approximated by a *propagator*: a contracting function that removes from the search space some or all non-solutions that violate the constraint. The greatest common fixpoint of the set of propagators yields an over-approximation of the set of all solutions; calculation of this fixpoint is interleaved with a backtracking search.

Constraint Programming with Strings

Apart from being “other things besides number[s],” what exactly are *strings*?

Strings are related to the informal concept of *words*; specifically, a string is a finite sequence of characters selected from an *alphabet* of symbols. For example, an alphabet could be the usual symbols:

$$\{a, b, c, d, \dots, z, \text{å}, \text{ä}, \text{ö}\}$$

Strings over this alphabet are words like ‘plantskolan’, ‘nödtvång’, or ‘genomsökning’, where the first character of the string ‘nödtvång’ is the symbol n, the second character is the symbol ö, and so forth. The set $\{1, \dots, 9\}$ could also be used as an alphabet, in which case strings would look like ‘42’ or ‘1520’. An alphabet might even include symbols for punctuation and white space, in which case this sentence, indeed this entire dissertation, is a string.

A CSP can have unknowns that are strings. In such a problem, there are string variables, which have domains that are sets of strings, and which are subject to constraints over strings. Constraints over strings are found in a wide variety of real-world problems such as test-case generation, program analysis, model checking, and web security.

There has been a large amount of research in recent years into methods for solving CSPs that include string variables. The treatment of string variables in string constraint solvers, indeed in computer science in general, can be divided

into three basic approaches. For example, the ADA 2005 reference manual³ defines a string variable type as “a one-dimensional array type whose component type is a character type,” [95] and lists three different ways that a string variable can be implemented:

- a *fixed-length string*, which can only represent string values of one particular length;
- a *bounded-length string*, which can represent string values whose length does not exceed some stated maximum value; and
- an *unbounded-length string*, which can represent strings of any finite length (in the case of ADA, until memory is exhausted).

Applications of constraint programming to CSPs that include string variables can also be divided into fixed-length, bounded-length, and unbounded-length approaches.⁴

A fixed-length string unknown with a length of n can be naturally modeled in CP by a sequence of n variables. The first variable in the sequence represents the first character in the string, and the domain of that variable represents all the symbols that the first character in the string could be. A CP solver might not allow variables that take the same values as the alphabet of the unknown string, but this is not a serious restriction: one can map the alphabet to integer numbers, representing the symbol *a* by the integer 1, and *b* by 2, and so on, and then model the candidate symbols of a character by an integer variable.

The primary advantage of modeling string unknowns this way is that it takes full advantage of existing CP solvers, with their extensive collection of included constraints, and efficient and correct fixpoint algorithms and backtracking search engines. Many string constraint problems can be modeled using combinations of known and commonly available integer constraints. Fixed-length string constraint problems, then, offer a relatively straightforward solution process, using off-the-shelf CP solvers.

The disadvantage of modeling string unknowns this way is that deciding the exact length of the string unknowns before searching for a solution results in a very narrow interpretation of the CSP. In CSPs with a tight connection between the lengths of multiple string unknowns, it might be difficult or impossible to find solutions to a fixed-length approximation of the CSP. Thus, a fixed-length model imposes a relatively high burden on the modeler: the model has to be carefully constructed in order to assure correctness.

The approach of using several integer variables to model a single string unknown is not an option when dealing with finite-domain unknowns of unbounded but finite length. The domain of a string variable with no bound on its length potentially holds an infinite number of strings, while integer variables

³As the world’s first computer programmer, Ada Lovelace has garnered a fair amount of posthumous recognition; surely the most dubious is her role as namesake of the United States Department of Defense’s venerable programming language, ADA.

⁴A survey of current research into the solving of string constraints, not limited to constraint programming approaches, may be found in Section 7.3.

in finite domain CP solvers have finite domains. However, unbounded-length string variables can be added as a new kind of variable to an existing CP solver.

A CP solver that provides a string variable type, and corresponding propagators for constraints over that type, allows for more compact and understandable models of CSPs with string unknowns: each unknown string in the CSP is modeled by a single variable, of type string, and constraints over strings can be expressed directly. Reducing the number of variables and constraints in a model also reduces burdens on the CP solver, and can lead to more efficient solving. Of course, the addition of a new variable type to a CP solver entails other additions, as well, such as propagators for each constraint, and support for making decisions on the new kind of variable domain during search. There is a tradeoff. On the one hand, there is the cost of designing and implementing a new variable type, concomitant propagators, and support for search. Also, adding new features increases the size and complexity of the CP solvers themselves, potentially making the core goals of correctness and efficiency harder to achieve; therefore, features like new variable types make sense only if they address an actual need. On the other hand, the potential benefits for modeling and efficiency are significant.

For a string variable type that allows unbounded-length strings, there are compact representations of infinite sets of strings, which could be used as a domain representation. For example, the expression $(a + b)^*c$ can be used to denote the set of strings that start with zero or more characters that are either the symbol a or the symbol b , followed immediately by one character that is the symbol c ; that is, the following infinitely large set:

$$\{ 'c', 'ac', 'bc', 'aac', 'abc', 'bac', \dots \}$$

The expression $(a + b)^*c$ is called a *regular expression*.

Unfortunately, this sort of compact representation for sets of strings comes at the cost of efficiency. A propagator must perform operations on the domains of variables to infer what values may be removed, and operations on representations like regular expressions have a relatively high cost in terms of complexity. As a result, CP solver variable types for unbounded-length strings have so far proved to be impractical.

We believe that the potential benefits of adding a string variable type to CP solvers, for both modeling and performance, outweigh the costs. We choose to work within the context of finite-domain CP solvers, so that unbounded-length domain representations are not reasonable; at the same time, fixed-length domains are unconvincing, for the reasons listed above. Therefore, we have chosen the middle road: we set out to implement a new string variable type for domains of *bounded*-length strings.

Thesis Statement 1

A bounded-length string variable type is a worthwhile addition to the variable types already found in CP solvers, for the following reasons:

- The presence of bounded-length string variables and string constraints substantially eases the modeling of string CSPs.
- A well-designed implementation of a bounded-length string variable type can effectively balance model expressiveness with propagator efficiency.
- Bounded-length string variables can lead to substantial performance improvements for CP solvers on string CSPs.

Galois Connections and Propagation

The choice of representation for the domains of a bounded-length string variable type depends heavily on the nature of the interaction between the string variables and the propagators for string constraints. Investigating how those propagators and variables should interact leads to other questions. Two central questions get at how strong the propagator should be. How much reasoning should these propagators perform? And what is the expected effect on the domains of the variables? In CP, these closely linked questions are answered in terms of properties of propagators and domains: *completeness*, a property of a propagators, is the answer to the first, while *consistency*, a property of domains, is the answer to the second.

Precisely defining completeness and consistency for bounded-length string variables turns out to be challenging. To find an answer, we turn to a theory of propagation as the computation of a constraint on an abstraction of the domains of its variables. Propagators remove some values from domains of variables, values that will never be used in a solution that satisfies the constraint. The strongest possible propagator would remove all those impossible values, but this is too hard, so propagators are usually weaker, meaning that they remove only some of those values. We can think of a weaker propagator as working not on the actual domains of its variables, but instead on a simpler representation of those domains. Such a propagator calculates which values can be removed from the domains by transforming them into an approximation, thereby abstracting away some of their details and making the calculation itself simpler and, not incidentally, faster. The resulting approximation can then be transformed back into an actual domain, but with the loss of some precision: some impossible values that could have been removed by a calculation on the actual domain are left by the calculation on the approximation.

This pair of transformations *must* obey certain rules in order for the resulting propagation to be correct. But it turns out that, quite commonly, the transformations are also related to each other in a very specific way: namely, that they

form a Galois connection. Galois connections have the pleasant property that they are easy to define, yet have far reaching consequences.

To precisely define completeness and consistency for a domain representation for bounded-length strings, we develop a theoretical framework of propagation strength based on Galois connections. This framework turns out to be surprisingly useful. For example, bounded-length strings are just complicated enough that it is not always obvious how a domain and a propagator should properly interact, until one considers the Galois connection that defines the domain representation, at which point the whole situation becomes much clearer.

Thesis Statement 2

The completeness of propagators and the consistency of domains can be precisely defined in terms of a Galois connection between a set of domains and a set of domain over-approximations. These Galois-based consistencies:

- allow the derivation of a propagator algorithm directly from a constraint definition,
- make it easier to precisely express complicated consistencies, and
- clarify important properties of a domain representation that is based on the over-approximation.

Contributions

The dissertation proceeds in two parts. In the first, we develop a theoretical framework, based on Galois connections, of propagation strength. In the second, we use this framework in the design of an efficient string variable type for a CP solver.

Part I

The first part of this dissertation is a theoretical exploration of the relationship between Galois connections and propagation strength.

The theoretical framework introduced in Chapter 3 allows consistency and completeness to be defined precisely by a Galois connection. The generalized consistency defined in this chapter, called $\langle\alpha, \gamma\rangle$ -consistency (Section 3.1), not only defines conditions on $\langle\alpha, \gamma\rangle$ -consistent domains, but also allows one to *derive* a propagator that maintains a given $\langle\alpha, \gamma\rangle$ -consistency (Section 3.2). $\langle\alpha, \gamma\rangle$ -consistency extends prior work on abstraction-based consistency: there is no attempt in [46] to generalize the relationship between abstraction and consistency, and while [98] does provide a definition of consistency as the greatest fixpoint of a computation on a set of abstract domains, the absence of a Galois connection in the general case limits the utility of the definition.

In Chapter 4, a sequence of Galois connections, each on the domain of a single variable, is used to naturally define relaxations of domain consistency (Section 4.2).

In Chapter 5, bounds consistency is generalized to a consistency applying to any partially ordered abstraction, irrespective of the type of variables involved (Section 5.2). This definition subsumes other bounds consistencies, such as $\text{bounds}(\mathbb{Z})$ consistency for integer variables (Section 5.2.2) and subset bounds consistency for set variables (Section 5.2.1), and with trivial modification $\text{bounds}(\mathbb{R})$ consistency for integer variables (Section 5.3). The relationship of non-subsumed consistencies (specifically $\text{bounds}(\mathbb{D})$ consistency and range consistency for integer variables, which are based on abstractions that are not founded on partial orders) to our theoretical framework is explained in Section 5.3.

In Chapter 6, more complicated consistencies are constructed using the theoretical framework of Part I, applying either to the domains of different variable types, or to domains subject to different over-approximations, or both.

Part II

The second part of this dissertation puts the theoretical framework of Part I into practice as a central part of the development of a bounded-length string variable type for constraint programming. This new decision variable type is designed starting from first principles, in a process informed by the Galois-based framework of Part I.

In Chapter 8 a rigorous over-approximation for sets of bounded-length strings is defined, and used as a representation for the domain of a string variable.

In Chapter 9, a propagator function, derived using the Galois-based framework, for this string representation is *translated* (Section 9.2.2) into a *propagator description*. This allows derivation, which applies to *any* $\langle\alpha, \gamma\rangle$ -consistency, to be used in place of the automated synthesis described in [71], which is limited to domain reasoning and $\text{bounds}(\mathbb{Z})$ reasoning. The translation is straightforward and could, in principle, be automated. The resulting propagator description may then be used to *generate* a solver-specific *propagator algorithm*, as described in [69].

The bounded-length string representation of Chapter 8 serves as an exemplar of a structured type that can be implemented as an *aggregation* of simple, built-in variable types, using methods I described in [VI]. The process, detailed in Chapter 9, applies to representations of structured variable types (e.g., sets, tuples, strings, graphs, etc.) in general, allowing the use of the above-mentioned derivation, translation, and code generation for an implementation of the structured type.

Finally, Chapter 10 describes the design and implementation of a bounded-length string variable type for CP solvers. The design choices made during this implementation are motivated by considering the role of variables in CP solvers, issues specific to string variables, and insights into the bounded-length

over-approximation, gleaned from the Galois-based framework. The performance of the implementation is experimentally evaluated in Chapter 11.

How to Read This Thesis

My publications are listed on page 190, and are referred to in the dissertation with roman numerals, i.e., [VI].

Sources

The following works were consulted during the preparation of this introduction. They also generally make for delightful reading.

1. Gleick, James *The Information: A History, a Theory, a Flood*. London: Fourth Estate, 2011.
2. Herzberg, Agnes M. and Murty, M. Ram . “Sudoku squares and chromatic polynomials.” In: *Notices of the AMS* 54, no. 6 (2007): 708–717.
3. Padua, Sydney. *The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer*. New York: Pantheon, 2015.
4. Wolfram, Stephen. “Untangling the Tale of Ada Lovelace”. (blog), 10 December 2015, blog.stephenwolfram.com/2015/12/untangling-the-tale-of-ada-lovelace/.

1. Constraint Programming: An Overview

For solving a *combinatorial problem*, one must find values for some *variables* within their given *initial domains* (which are discrete sets of candidate values), so that all given *constraints* (which are relations) on these variables are satisfied. One distinguishes between *combinatorial optimization problems*, where some numeric cost expression, called the *objective function*, on these variables is minimized, and *combinatorial satisfaction problems*, where there is no objective function.¹

1.1 Constraint Problems

Many combinatorial satisfaction problems may be formally expressed as a *constraint satisfaction problem* (CSP), which is a tuple $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$ with tuple $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ of *variables*, set $\overline{\mathbf{D}} = \overline{D}_1 \times \dots \times \overline{D}_n$ of *candidate solutions* such that every X_i is required to take a value in $\overline{D}_i$, and set $\mathbf{C} = \{C_1, \dots, C_t\}$ of *constraints*. Each constraint C_j is a relation on ground instances of the variables $\mathbf{X}$. An *assignment* is a tuple $\langle a_1, \dots, a_n \rangle \in \overline{\mathbf{D}}$, so for every $X_i \in \mathbf{X}$ we have $a_i \in \overline{D}_i$. An assignment $\langle a_1, \dots, a_n \rangle$ is a *solution* to the CSP if it *satisfies* every constraint, i.e., if $\langle a_1, \dots, a_n \rangle \in C_j$ for every $1 \leq j \leq t$. The satisfiability of a constraint typically does not depend upon the values of all the variables in $\mathbf{X}$; for a constraint C_j , the tuple S_j , called the *scope* of C_j , consists of those variables that are relevant to the constraint, that is every variable X_i for which there exist a pair of assignments $a = \langle a_1, \dots, a_n \rangle$ and $a' = \langle a'_1, \dots, a'_n \rangle$ such that a satisfies C_j and a' does not, where $a_i \neq a'_i$ and $a_k = a'_k$ for all $k \in [1, n]$ such that $k \neq i$. We call the cardinality of S_j the *arity* of C_j . Except where otherwise noted, we assume without loss of generality² that the scope of each constraint is equal to $\mathbf{X}$. In a similar fashion, a combinatorial optimization problem may be expressed as a *constrained optimization problem* (COP); the concepts and methods we discuss in sequel apply to both CSPs and COPs without modification, so henceforth we refer solely to the former.

¹Naturally, the objective function, when present, may equally be a benefit expression to be maximized.

²This simplifying assumption is reasonable only because we generally treat constraints in isolation. Expressing a conjunction of constraints is simpler with the more common definition of a constraint as a pair consisting of a scope, which is a tuple of variables, and a relation over the Cartesian product of the domains of the variables in the scope; an assignment, then, is a map from variables to values. Neither of these conventions is required for clarity in the present work, so we simplify notation by omitting them. Expansion of our treatment to the standard model is straightforward, if occasionally unwieldy.

1.2 Constraint Programming Solvers

Constraint programming (CP) is an approach to the modeling and solving of combinatorial problems. Its core idea is to encapsulate islands of structure that are commonly occurring within combinatorial problems in *constraints*. Beside the usual binary comparison constraints ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$), with arguments involving the usual binary arithmetic operators ($+$, $-$, etc.) applied to numeric constants and variables, there are *combinatorial constraints*, involving more complex structures and a non-fixed number of not necessarily numeric arguments. For example, the $\text{ALLDIFFERENT}(X_1, \dots, X_k)$ constraint [82] requires its $k \leq n$ variables X_i to take pairwise different values: it wraps the conjunction of $\frac{n \cdot (n-1)}{2}$ constraints $X_i \neq X_j$ of arity 2 into a single n -ary constraint. Hundreds of useful constraints are described in the *Global Constraint Catalogue* [10]. Constraints can be combined to *model* a combinatorial problem in a declarative and very high-level fashion, but they can also be exploited during the solution process.

A CP solver uses systematic tree search, together with backtracking, to solve combinatorial problems, and performs at every node of the search tree a particular kind of inference called *propagation*, the aim of which is to shrink the search space.

Domains

A *domain* is a set of assignments:

Definition 1.1: Domain

For a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$, set $\overline{\mathbf{D}} = \overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n$ of candidate solutions, and constraints $\mathbf{C}$, a *domain* $\mathcal{D}$ is a subset of $\overline{\mathbf{D}}$.

For a domain $\mathcal{D}$, the function $\text{dom}(X_i)$ is the projection of $\mathcal{D}$ onto the set of values at the i -th index, i.e., the set of all values taken by variable X_i in any assignment in $\mathcal{D}$. We call $\text{dom}(X_i)$ the *variable domain* of X_i .

In practice, CP solvers do not represent a domain $\mathcal{D}$ extensionally, but rather store and reason on an over-approximation thereof, most typically the Cartesian closure $\text{dom}(X_1) \times \dots \times \text{dom}(X_n)$, so that only the n sets $\text{dom}(X_i)$ are represented. In part this decision is motivated by the fact that an extensional representation of $\mathcal{D}$ requires space exponential in the number n of variables, but additionally an extensional representation conceals much of the combinatorial structure of the problem [86]. So pivotal is this design decision that theoretical treatments of CP commonly define a domain as a Cartesian closure (e.g., [13]). In our framework, we take the more general view that a domain can be any set of assignments, not necessarily built from the domains of individual variables; a CP solver, then, reasons on some over-approximation of a domain (as defined, in our case, by a Galois connection). As a result, our framework

Figure 1.1. Creation of a propagator algorithm that maintains a particular consistency, via two pathways: either a propagator function is derived from the constraint semantics and then translated into an indexical description, or the constraint semantics is translated into the indexical language and then used to synthesize the indexical description. Either way, automated code generation starting from the resulting indexical description yields a solver-specific propagator algorithm.

encompasses traditional Cartesian domain approximations, but also allows for extension to recent work on non-Cartesian domain approximations, such as decision diagrams [4, 49] or tuples [56, 70, 79].

Propagation

A *propagator* is a partial implementation of a constraint: it removes from a domain some (but not necessarily all) tuples that do not satisfy the constraint. The term *propagator* is used in CP at several levels of abstraction. To minimize confusion, we distinguish between the following concepts, as illustrated also in Figure 1.1:

- *propagator function*: a contracting function on domains, defining the effect of propagating the constraint on a domain,
- *propagator description*: a high-level, declarative, solver-independent description of a propagator as a conjunction of range restriction operations, written for example in an *indexical language* [101],
- *propagator algorithm*: a software entity written for a specific CP solver, encapsulating a procedural inference algorithm that prunes a domain.

In a CP solver, each individual constraint definition is implemented by one or more propagator algorithms that prune assignments from a domain that do not satisfy the constraint. A propagator algorithm implementing a constraint only reasons *locally*, namely about the variables in the scope of that constraint, rather than globally, about all the variables of the problem.

Example 1.1

Consider the constraint $\text{ALLDIFFERENT}(X_1, X_2, X_3)$, and the domain

$$\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}$$

The strongest possible propagation algorithm for this constraint would return a domain consisting solely of those four tuples that satisfy the constraint, namely

$$\{\langle 2, 1, 3 \rangle, \langle 2, 3, 1 \rangle, \langle 4, 1, 3 \rangle, \langle 4, 3, 1 \rangle\}$$

However, propagating ALLDIFFERENT by considering every tuple in a domain is computationally expensive in the general case.

When pruning a domain, there is *no* obligation for a propagator algorithm to remove all assignments in the domain that do not satisfy the implemented constraint. A constraint may be implemented by several propagator algorithms, which may be distinguished according to the *strength* of the pruning they achieve. In practice, the choice of propagator strength is motivated by efficiency, either of time (e.g., using a propagator algorithm that removes fewer impossible domain elements but is computationally less expensive) or of space (e.g., using a domain representation that over-approximates the domain but is more compact than an exact representation). In many cases this choice may be made at the modeling level, allowing the modeler to select a tradeoff between strength and efficiency that is appropriate for a given problem. The strength of a propagation algorithm is typically distinguished according to its achieved *consistency*. For instance, under *domain consistency* every value of every variable domain participates in some assignment satisfying the constraint involving values in the variable domains of the other variables.

Example 1.2

Continuing from Example 1.1, when starting from the domain

$$\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}$$

propagation to domain consistency leads to the domain

$$\{2, 4\} \times \{1, 3\} \times \{1, 3\}$$

since X_2 and X_3 must split the values 1 and 3 between themselves so that X_1 cannot take either of these two values. Propagation to domain consistency prunes half of the $4 \cdot 2 \cdot 4 = 16$ assignments that were present in the original domain; however, half of the remaining $2 \cdot 2 \cdot 2 = 8$ assignments, such as $\langle 2, 1, 1 \rangle$, do not satisfy the constraint ALLDIFFERENT .

A propagator algorithm *is* obligated to correctly distinguish between an assignment that is a solution, and one that is a non-solution. When it can be determined that no candidate solution in the domain satisfies the constraint, a propagator algorithm reports a *failure*, and when it can be determined that all remaining candidate solutions satisfy the constraint, a propagator algorithm reports *subsumption* (or *entailment*).

Example 1.3

For the constraint $\text{ALLDIFFERENT}(X_1, X_2, X_3)$, starting from the domain

$$\{1, 2\} \times \{1, 2\} \times \{1, 2\}$$

propagation to domain consistency leads to the propagator algorithm signalling failure, because it is impossible to assign two values to three variables so that the latter are pairwise distinct. From the domain

$$\{1\} \times \{3\} \times \{2, 4\}$$

propagation to domain consistency leads to no pruning, but the propagator algorithm can simultaneously detect that all remaining $1 \cdot 1 \cdot 2 = 2$ candidate solutions actually are solutions, so that the propagator algorithm can signal subsumption of its constraint.

Fixpoint Algorithm

For a conjunction of constraints, each constraint is propagated to its desired consistency: this may take multiple invocations of some propagator algorithms and is choreographed by a fixpoint algorithm, which computes the greatest common fixpoint of all the propagator algorithms.

Example 1.4

Consider the conjunction

$$X_1 \neq X_2 \wedge X_2 \neq X_3 \wedge X_3 \neq X_1$$

of binary disequality constraints, which is equivalent to the combinatorial constraint $\text{ALLDIFFERENT}(X_1, X_2, X_3)$. Starting from the domain

$$\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}$$

propagation to the greatest common fixpoint of the three propagator algorithms, each propagated to domain consistency, achieves no pruning at all (unlike in Example 1.2), because each individual $\alpha \neq \beta$ constraint is unsatisfiable only when its two variables α and β range over the same

singleton domain. From the domain

$$\{1, 2\} \times \{1, 2\} \times \{1, 2\}$$

propagation to domain consistency also achieves no pruning at all (unlike in Example 1.3), so that search (see below) is necessary to reveal the unsatisfiability of the conjunction.

Example 1.4 not only exemplifies the fixpoint algorithm, but also justifies the existence of combinatorial constraints such as `ALLDIFFERENT`.

There exists a substantial literature on propagator fixpoint algorithms (see [13] for a survey); however, fixpoint calculation in CP is agnostic with respect to propagator algorithms, and hence falls outside the scope of the current work.

Search

CP solving is an interleaving of propagator fixpoint computation and systematic tree search. This is illustrated by Algorithm 1.1: the recursive algorithm `search` takes as arguments a domain $\mathcal{D}$ and a set $\mathcal{P}$ of propagator algorithms, and returns the set of all assignments in $\mathcal{D}$ that satisfy all the constraints implemented by the propagators in $\mathcal{P}$; in other words, `search` returns the set of all solutions in $\mathcal{D}$. The domain at each search tree node is the greatest common fixpoint, calculated by `propagate` (line 1), of all the propagator algorithms in $\mathcal{P}$. At the root of the tree, the domain $\mathcal{D}$ is the greatest common fixpoint that is reached starting from the set of all candidate solutions $\overline{\mathbf{D}}$. The tree is grown at some leaf by partitioning the domain to create $m \geq 2$ sub-problems, obtained by mutually exclusive *decisions* on some variables. The decisions are chosen by `branch` (line 5), a *branching heuristic*, which is either the default of the CP solver or provided by the human modeler of the combinatorial problem. Decisions must be constraints themselves, hence they have propagator algorithms (e.g., $p_1, \dots, p_m$): the fixpoint algorithm `propagate` is launched for computing each child of the considered node (line 6). Backtracking is performed whenever the fixpoint algorithm fails because the propagator algorithm

Function `search` ($\mathcal{D}, \mathcal{P}$)

```

1   $\mathcal{D}' \leftarrow \text{propagate}(\mathcal{D}, \mathcal{P});$ 
2  if  $\mathcal{D}'$  is failed then
3    return  $\emptyset$ 
4  if  $|\mathcal{D}'| > 1$  then
5     $\langle p_1, \dots, p_m \rangle \leftarrow \text{branch}(\mathcal{D}')$ ;
6    return  $\text{search}(\mathcal{D}', \mathcal{P} \cup \{p_1\}) \cup \dots \cup \text{search}(\mathcal{D}', \mathcal{P} \cup \{p_m\})$ 
7  else
8    return  $\mathcal{D}'$ 
```

Algorithm 1.1: CP solving as an interleaving of propagation and search.

of some constraint signals failure (line 2). When, after the fixpoint calculation at a given node, the domain $\mathcal{D}'$ consists of a single assignment, which must therefore be a solution, it is returned (line 8).

The algorithm `search` is highly simplified, and would in practice be subject to a large number of optimizations, such as for example dynamically reducing the size of $\mathcal{P}$ by removing propagator algorithms corresponding to subsumed constraints. For an overview of practical issues in CP solver design, see [86].

Part I: Abstract Consistency

If one recognizes that a Galois connection is involved in a phenomenon that may be relatively complex, then many aspects of that phenomenon immediately become clear; and thus, the whole situation typically becomes much easier to understand.

“A Primer on Galois Connections”
MARCEL ERNÉ et al.

Abstract interpretation (AI) is a widely applicable theory of approximation, of particular interest in program analysis and verification: it provides a methodology for the systematic construction of a sound over-approximation for a problem that is itself too complex to admit efficient solution. AI models a problem as two ordered sets of states: a set of *concrete states*, corresponding to actual problem states, and a set of *abstract states* over-approximating the concrete states. The over-approximation (i.e., the relationship between the two sets of states) is defined by two functions: an *abstraction function* that maps concrete states to abstract states, and a *concretization function* that maps abstract states to concrete states. A wide variety of approximations are detailed in the literature of AI; they may be combined in order to create, for a wide variety of problems, abstract semantics possessing a desirable balance of computational cost and precision.

AI functions as a general theory of abstraction, leveraging results from lattice theory and, most importantly, the notion of *Galois connections*, a notion that is more easily formally defined (see Definition 2.8) than informally described; nevertheless, it is not too far off the mark to say that, taken together, the abstraction function and concretization function of the previous paragraph are a Galois connection, as long as both functions preserve the orders of the two sets of states. The concept of Galois connections is engagingly simple, yet leads to a wealth of results: a pleasant situation, indeed.

Expressing CP in terminology inspired by AI, a constraint is a contracting operation on the set of concrete domains, while the corresponding propagator is a sound approximation of that constraint on the set of abstractions of those domains. This set of abstractions, together with the Galois connection it shares with the set of concrete domains, directly corresponds to how strongly the propagator approximates the constraint; i.e., using the terminology of CP, this Galois connection uniquely defines the *consistency* maintained by the propagator.

Part I explores the relationship between consistency and Galois connections in detail. The theoretical framework that emerges from this exploration is both general enough to precisely describe consistencies encompassing a wide variety of propagation strengths and variable types, and specific enough to have practical applications. Indeed, Part II, which describes the design and implementation of a new type of string variable, can in part be read as an extended case study in the application of the theoretical framework developed in Part I.

* * *

Part I proceeds as follows:

- Chapter 2 provides a summary of the relevant background from lattice theory and AI, formalizes the notions of propagators, consistency, and completeness, and describes an idealized *exact consistency*, then surveys the theoretical work connecting CP and AI, and its relationship to the framework developed in this thesis.

- Chapter 3 lays out the heart of the theoretical framework of Part I: a general theorem that defines the consistency, called $\langle \alpha, \gamma \rangle$ -consistency, that is induced by an arbitrary domain abstraction.
- Chapter 4 applies $\langle \alpha, \gamma \rangle$ -consistency to domains that are approximated by a Cartesian product, deriving propagators that maintain domain consistency.
- Chapter 5 applies $\langle \alpha, \gamma \rangle$ -consistency to domains approximated by an interval, giving a generalized definition of bounds consistency that subsumes the most common definitions of bounds consistency (specifically, $\text{bounds}(\mathbb{Z})$ consistency and $\text{bounds}(\mathbb{R})$ consistency for integers, and subset bounds consistency for sets).
- Chapter 6 applies $\langle \alpha, \gamma \rangle$ -consistency to heterogeneous consistencies that involve multiple variable types, multiple domain approximations, or both.

2. Preliminaries

Before the theoretical framework outlined in Section 2.4 can be further developed, some additional groundwork needs to be laid.

For the reader unfamiliar with the basic concepts of lattice theory, Galois connections, or abstract interpretation, a brief overview of the relevant background material is provided in Section 2.1.

In Section 2.2, propagator functions, which were described in Section 1.2 as contracting functions that partially implement constraints, are given a formal definition. In Section 2.2.1 two important properties, consistency and completeness, are introduced. These properties describe the strength of a propagator function. In Section 2.2.2 the consistency associated with the strongest possible propagator functions, *exact consistency*, is defined. In fact, exact consistency is *too* strong to be practical; it will, however, be used in subsequent chapters when defining other consistencies that are more practical.

There follows, in Section 2.3, a survey of prior work on the connection between constraint programming and abstract interpretation. In particular, two viewpoints of propagation are discussed: from the first viewpoint, propagation is a computation on an abstract domain, and from the second viewpoint, propagation is a computation bracketed by closure operations. Section 2.4 then draws upon aspects of each of these two viewpoints to provide a sketch of the theoretical framework developed in the remainder of Part I, which characterizes propagation in terms of Galois connections.

2.1 Lattice Theory and Galois Connections

We briefly introduce concepts from lattice theory and *abstract interpretation* (AI), which we use in sequel to express formally the relationship between propagators and constraints. Proofs of the lemmas presented in this section are easily obtained (from, e.g., [29]), so they are omitted here.

Notation

We denote sets in uppercase script (e.g., $\mathcal{A}$, $\mathcal{B}$, etc.) and scalars in lowercase (e.g., a , b , etc.). We refer to the set $\{\ell, \ell + 1, \dots, u - 1, u\}$ of integers, which is the empty set $\emptyset$ if $\ell > u$, using the notation $[\ell, u]$. For a set $\mathcal{A}$, we write $\wp(\mathcal{A})$ for its powerset, and $|\mathcal{A}|$ for its cardinality. We use angled brackets to denote an ordered sequence, or tuple (e.g., $\langle a_1, \dots, a_n \rangle$); however, we follow

the convention of omitting the brackets for a tuple that is the sole argument of a function, writing $f(a, b)$ instead of $f(\langle a, b \rangle)$.

For sets $\mathcal{A}$ and $\mathcal{B}$, we write $\mathcal{A} \times \mathcal{B}$ for the Cartesian product $\{\langle a, b \rangle \mid a \in \mathcal{A} \wedge b \in \mathcal{B}\}$; for a set $\mathcal{P}$ of sets indexed by the set $\mathcal{I} = [1, n]$, we write $\prod_{i \in \mathcal{I}} \mathcal{P}_i$ for the Cartesian product $\mathcal{P}_1 \times \cdots \times \mathcal{P}_n$. We further lift the concept of Cartesian product to functions: for a pair of functions $f: \mathcal{P} \rightarrow \mathcal{Q}$ and $g: \mathcal{R} \rightarrow \mathcal{S}$, the Cartesian product of f and g , written $f \times g$, is the function

$$f \times g: \mathcal{P} \times \mathcal{R} \rightarrow \mathcal{Q} \times \mathcal{S} \text{ where } (f \times g)(p, r) = \langle f(p), g(r) \rangle$$

Similarly, for a set of functions $\mathcal{F}$ indexed by a set $\mathcal{I} = [1, n]$, where $f_i: \mathcal{P}_i \rightarrow \mathcal{Q}_i$ is in $\mathcal{F}$ for all $i \in \mathcal{I}$, we write $\prod_{i \in \mathcal{I}} f_i$ for the Cartesian product $f_1 \times \cdots \times f_n$.

2.1.1 Partially Ordered Sets

An ordered set is a set whose elements are distinguished by a binary relation called an *order*.

Definition 2.1: Poset

A *partially ordered set*, or *poset*, is a pair $\langle \mathcal{P}, \leq \rangle$ where $\mathcal{P}$ is a set and $\leq$ is a binary relation on $\mathcal{P}$ that is reflexive, anti-symmetric, and transitive. If all elements of $\mathcal{P}$ are comparable (i.e., either $p_1 \leq p_2$ or $p_2 \leq p_1$ for all $p_1, p_2 \in \mathcal{P}$), then $\mathcal{P}$ is a *totally ordered set*.

As an example, the powerset $\wp(\mathcal{X})$ of any set $\mathcal{X}$ is a poset under subset inclusion, written $\langle \wp(\mathcal{X}), \subseteq \rangle$.

Definition 2.2: Upper and Lower Bounds

Let $\langle \mathcal{P}, \leq \rangle$ be a poset. An *upper bound* of $\mathcal{S} \subseteq \mathcal{P}$ is an $x \in \mathcal{P}$ such that for all $s \in \mathcal{S}$, we have $s \leq x$. The *least upper bound (lub)* of $\mathcal{S}$, denoted $\bigvee \mathcal{S}$, is an upper bound of $\mathcal{S}$ such that for every $y \in \mathcal{P}$ that is an upper bound of $\mathcal{S}$, we have $\bigvee \mathcal{S} \leq y$. Similarly, a *lower bound* of $\mathcal{S} \subseteq \mathcal{P}$ is an $x \in \mathcal{P}$ such that for all $s \in \mathcal{S}$, we have $x \leq s$, and the *greatest lower bound (glb)* of $\mathcal{S}$, denoted $\bigwedge \mathcal{S}$, is a lower bound of $\mathcal{S}$ such that for every $y \in \mathcal{P}$ that is a lower bound of $\mathcal{S}$, we have $y \leq \bigwedge \mathcal{S}$.

The Cartesian product of sets may be extended to posets in a natural manner:

Lemma 2.3: Cartesian Product of Posets

Let $\langle \mathcal{P}_1, \leq^1 \rangle, \dots, \langle \mathcal{P}_n, \leq^n \rangle$ be posets. The Cartesian product $\mathcal{P}_1 \times \dots \times \mathcal{P}_n$ is a poset under the component-wise order defined by:

$$\langle x_1, \dots, x_n \rangle \leq \langle y_1, \dots, y_n \rangle \equiv \forall i \in [1, n] : x_i \leq^i y_i$$

2.1.2 Lattices

A *lattice* is a poset for which the glb and lub exist for every pair of elements:

Definition 2.4: Lattice

Let $\langle \mathcal{P}, \leq \rangle$ be a poset.

- (i) If $\bigvee \{x, y\}$ and $\bigwedge \{x, y\}$ exist for all $x, y \in \mathcal{P}$, then $\mathcal{P}$ is called a *lattice*.
- (ii) If $\bigvee S$ and $\bigwedge S$ exist for all $S \subseteq \mathcal{P}$, then $\mathcal{P}$ is called a *complete lattice*.

We do not require a lattice to have either a maximum or minimum element, however any complete lattice must possess both.

As an example, under their usual numeric orders, both $\mathbb{Z}$ and $\mathbb{R}$ are lattices but not complete lattices. Consider a set $S \subseteq \mathbb{Z}$: the lub of S is the maximum element of S , which exists only if S is finite. On the other hand, for any set $\mathcal{X}$, the poset $\langle \wp(\mathcal{X}), \subseteq \rangle$ is a complete lattice where the lub is the union and the glb the intersection. In this case $\mathcal{X}$ is the top element, $\emptyset$ is the bottom element, and both bounds exist for all $S \subseteq \wp(\mathcal{X})$: if S is infinite, then $\bigvee S = \mathcal{X}$, while even if S contains non-overlapping sets we still have $\bigwedge S = \emptyset$. We write this complete lattice as $\langle \wp(\mathcal{X}), \subseteq, \cup, \cap \rangle$, and call it the *powerset lattice* of $\mathcal{X}$.

Lattices are closed under the Cartesian product operation, with the glb and lub of the resulting lattice defined component-wise:

Lemma 2.5

Let $\langle \mathcal{P}_1, \leq^1, \bigvee^1, \bigwedge^1 \rangle, \dots, \langle \mathcal{P}_n, \leq^n, \bigvee^n, \bigwedge^n \rangle$ be lattices. The Cartesian product $\mathcal{P}_1 \times \dots \times \mathcal{P}_n$ is a lattice with the glb and lub of a set of tuples $\mathcal{T} \subseteq \mathcal{P}_1 \times \dots \times \mathcal{P}_n$ defined component-wise as:

$$\begin{aligned} \bigvee \mathcal{T} &= \left\langle \bigvee^1 \{t_1 \in \mathcal{P}_1 \mid \langle t_1, \dots, t_n \rangle \in \mathcal{T}\}, \right. \\ &\quad \left. \dots, \bigvee^n \{t_n \in \mathcal{P}_n \mid \langle t_1, \dots, t_n \rangle \in \mathcal{T}\} \right\rangle \\ \bigwedge \mathcal{T} &= \left\langle \bigwedge^1 \{t_1 \in \mathcal{P}_1 \mid \langle t_1, \dots, t_n \rangle \in \mathcal{T}\}, \right. \\ &\quad \left. \dots, \bigwedge^n \{t_n \in \mathcal{P}_n \mid \langle t_1, \dots, t_n \rangle \in \mathcal{T}\} \right\rangle \end{aligned}$$

We omit the ordering relation and bound operators where clarity allows, referring then to a lattice $\langle \mathcal{P}, \leq, \bigvee, \bigwedge \rangle$ as $\mathcal{P}$.

2.1.3 Closure and Reduction Operators

Definition 2.6: Closure Operator

Let $\langle \mathcal{P}, \leq \rangle$ be a poset. A function $\phi : \mathcal{P} \rightarrow \mathcal{P}$ is called a *closure operator* on $\mathcal{P}$ if, for all $x, y \in \mathcal{P}$, the function ϕ is:

- (i) *extensive*: $x \leq \phi(x)$
- (ii) *monotonic*: $x \leq y \implies \phi(x) \leq \phi(y)$
- (iii) *idempotent*: $\phi(\phi(x)) = \phi(x)$

An element $p \in \mathcal{P}$ is said to be *closed* under ϕ (or ϕ -*closed*) if and only if $\phi(p) = p$.

An example of a closure operator is the interval closure for integer sets, which maps a set $\mathcal{S} \in \wp(\mathbb{Z})$ to the set $\{i \in \mathbb{Z} \mid \min(\mathcal{S}) \leq i \leq \max(\mathcal{S})\}$. Replacing the extensivity property of Definition 2.6 with its dual yields the definition of a reduction operator: Similarly, a reduction operator is monotonic and idempotent in the sense of Definition 2.6, but satisfies the dual of the extensivity property:

Definition 2.7: Reduction Operator

Let $\langle \mathcal{P}, \leq \rangle$ be a poset. A function $\phi : \mathcal{P} \rightarrow \mathcal{P}$ is called a *reduction operator* on $\mathcal{P}$ if ϕ is monotonic and idempotent (in the sense of Definition 2.6), and also *contracting*, meaning that $\phi(x) \leq x$ for all $x \in \mathcal{P}$. An element $p \in \mathcal{P}$ is said to be *reduced* under ϕ (or ϕ -*reduced*) if and only if $\phi(p) = p$.

2.1.4 Galois Connections

As a framework for describing abstractions of domains, we will rely upon the notion of Galois connections:

Definition 2.8: Galois Connection

If $\langle \mathcal{P}, \leq, \bigvee, \bigwedge \rangle$ and $\langle \mathcal{P}^\#, \leq^\#, \bigvee^\#, \bigwedge^\# \rangle$ are complete lattices, then a pair of functions $\alpha : \mathcal{P} \rightarrow \mathcal{P}^\#$ and $\gamma : \mathcal{P}^\# \rightarrow \mathcal{P}$ is a *Galois connection* if and only if

$$\forall x \in \mathcal{P}, \forall y \in \mathcal{P}^\# : \alpha(x) \leq^\# y \Leftrightarrow x \leq \gamma(y)$$

We denote this Galois connection as

$$\langle \mathcal{P}, \leq, \bigvee, \bigwedge \rangle \xleftrightarrow[\alpha]{\gamma} \langle \mathcal{P}^\#, \leq^\#, \bigvee^\#, \bigwedge^\# \rangle$$

The definition of Galois connections exists in two versions: the original version by Galois was *antitone*, but in computer science the more common version, given above, is *monotone*, as it possesses the monotonicity property in the following lemma:

Lemma 2.9

Assume that

$$\langle \mathcal{P}, \leq, \vee, \wedge \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \mathcal{P}^\#, \leq^\#, \vee^\#, \wedge^\# \rangle$$

is a Galois connection between posets $\mathcal{P}$ and $\mathcal{P}^\#$. Let $p, p_1, p_2 \in \mathcal{P}$ and $p^\#, p_1^\#, p_2^\# \in \mathcal{P}^\#$. Then the following properties hold:

(i) The function $\gamma \circ \alpha$ is *extensive*, and the function $\alpha \circ \gamma$ is *contracting*:

$$p \leq (\gamma \circ \alpha)(p) \quad \text{and} \quad (\alpha \circ \gamma)(p^\#) \leq^\# p^\#$$

(ii) Both α and γ are *monotonic*:

$$\begin{aligned} p_1 \leq p_2 &\implies \alpha(p_1) \leq^\# \alpha(p_2) \\ p_1^\# \leq^\# p_2^\# &\implies \gamma(p_1^\#) \leq \gamma(p_2^\#) \end{aligned}$$

(iii) The functions α and γ are *quasi-inverses*:

$$(\alpha \circ \gamma \circ \alpha)(p) = \alpha(p) \quad \text{and} \quad (\gamma \circ \alpha \circ \gamma)(p^\#) = \gamma(p^\#)$$

The pair of functions in any Galois connection uniquely determine one another:

Lemma 2.10

Given any Galois connection $\langle \mathcal{P}, \leq, \vee, \wedge \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \mathcal{P}^\#, \leq^\#, \vee^\#, \wedge^\# \rangle$, the functions α and γ are given by the following formulae:

$$\begin{aligned} \alpha(p) &= \bigwedge^\# \{ p^\# \in \mathcal{P}^\# \mid p \leq \gamma(p^\#) \} \\ \gamma(p^\#) &= \bigvee \{ p \in \mathcal{P} \mid \alpha(p) \leq^\# p^\# \} \end{aligned}$$

A Galois connection between two posets defines the conditions under which lubs and glbs are coexistent in the posets. The *preservation* of lubs and glbs between posets is defined as follows:

Definition 2.11: Preservation of Existing Bounds

Let $\mathcal{P}$ and $\mathcal{P}^\sharp$ be posets, $\mathcal{S} \subseteq \mathcal{P}$ a set, and $\phi: \mathcal{P} \rightarrow \mathcal{P}^\sharp$ a function. We say that ϕ *preserves existing lubs* if whenever $\bigvee \mathcal{S}$ exists in $\mathcal{P}$ then $\bigvee^\sharp \{p^\sharp \in \mathcal{P}^\sharp \mid \exists s \in \mathcal{S} : p^\sharp = \phi(s)\}$ exists in $\mathcal{P}^\sharp$ and is equal to $\phi(\bigvee \mathcal{S})$. Preservation of existing glbs is defined dually.

The relationship between preservation of bounds and Galois connections is a special case of the adjoint functor theorem from category theory [53]:

Lemma 2.12: Adjoint Functor Theorem for Complete Lattices

Let $\mathcal{P}$ and $\mathcal{Q}$ be posets, and $\phi: \mathcal{P} \rightarrow \mathcal{Q}$ a function.

- (i) If $\mathcal{P}$ is a complete lattice, then ϕ preserves existing glbs if and only if ϕ possesses an *upper adjoint* $\phi^\sharp: \mathcal{P} \xleftarrow[\phi]{\phi^\sharp} \mathcal{Q}$ is a Galois connection.
- (ii) If $\mathcal{Q}$ is a complete lattice, then ϕ preserves existing lubs if and only if ϕ possesses a *lower adjoint* $\phi^\flat: \mathcal{Q} \xleftarrow[\phi^\flat]{\phi} \mathcal{P}$ is a Galois connection.

Galois connections may be constructed as combinations of other Galois connections. Many such construction methods are detailed in the literature (see, e.g., [73]); the following lemma presents in particular three methods that are used in this thesis:

Lemma 2.13: Construction of Galois Connections

Galois connections may be constructed by combining other Galois connections as follows:

- (i) The *composition* of Galois connections $\mathcal{P} \xleftarrow[\alpha_1]{\gamma_1} \mathcal{Q}$ and $\mathcal{Q} \xleftarrow[\alpha_2]{\gamma_2} \mathcal{R}$ is the Galois connection $\mathcal{P} \xleftarrow[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2} \mathcal{R}$, sometimes written

$$\mathcal{P} \xleftarrow[\alpha_1]{\gamma_1} \mathcal{Q} \xleftarrow[\alpha_2]{\gamma_2} \mathcal{R}$$

- (ii) The *Cartesian product* of Galois connections $\mathcal{P}_1 \xleftarrow[\alpha_1]{\gamma_1} \mathcal{Q}_1$ and $\mathcal{P}_2 \xleftarrow[\alpha_2]{\gamma_2} \mathcal{Q}_2$ is the Galois connection

$$\mathcal{P}_1 \times \mathcal{P}_2 \xleftarrow[\alpha_1 \times \alpha_2]{\gamma_1 \times \gamma_2} \mathcal{Q}_1 \times \mathcal{Q}_2$$

- (iii) The *direct product* of Galois connections $\mathcal{P} \xleftarrow[\alpha_1]{\gamma_1} \mathcal{Q}_1$ and $\mathcal{P} \xleftarrow[\alpha_2]{\gamma_2} \mathcal{Q}_2$ is the Galois connection

$$\mathcal{P} \xleftarrow[\alpha]{\gamma} \mathcal{Q}_1 \times \mathcal{Q}_2 \quad \text{where} \quad \begin{aligned} \alpha(p) &= \langle \alpha_1(p), \alpha_2(p) \rangle \\ \gamma(q_1, q_2) &= \gamma_1(q_1) \cap \gamma_2(q_2) \end{aligned}$$

Finally, we note that every Galois connection induces a closure operator:

Lemma 2.14: Induced Closure Operators

Any Galois connection $\mathcal{P} \xleftrightarrow[\alpha]{\gamma} \mathcal{P}^\#$ induces the closure operator $\gamma \circ \alpha : \mathcal{P} \rightarrow \mathcal{P}$ on $\mathcal{P}$.

Conversely, for every closure operator ϕ on a complete lattice $\mathcal{P}$ there exists a Galois connection $\mathcal{P} \xleftrightarrow[\alpha]{\gamma} \mathcal{P}^\#$ to some complete lattice $\mathcal{P}^\#$ such that ϕ is equal to $\gamma \circ \alpha$. One such Galois connection for ϕ is $\mathcal{P} \xleftrightarrow[\phi]{\text{id}^\mathcal{P}} \mathcal{P}_\phi$, where $\text{id}^\mathcal{P}$ is the identity function on $\mathcal{P}$ and $\mathcal{P}_\phi$ is the set of elements of $\mathcal{P}$ that are closed under ϕ ; other Galois connections would also induce ϕ .

2.1.5 Sound Approximation

In *abstract interpretation* (AI), the concept of abstraction is formalized using Galois connections. In AI, the states of the actual problem form a lattice, called the *concrete lattice*. These states are then related to a lattice of approximations, called the *abstract lattice*. The concrete lattice $\mathcal{P}$ and the abstract lattice $\mathcal{P}^\#$ form a Galois connection $\mathcal{P} \xleftrightarrow[\alpha]{\gamma} \mathcal{P}^\#$ as defined by the *abstraction function*, α , mapping states of the concrete lattice to states of the abstract lattice, and the *concretization function*, γ , mapping states in the opposite direction.

The following concepts from AI are used in Chapter 3 to construct a framework relating consistency to abstraction [28] (see also [46]):

Definition 2.15: Sound Approximation

Let $\langle \mathcal{P}, \leq, \vee, \wedge \rangle \xleftrightarrow[\alpha]{\gamma} \langle \mathcal{P}^\#, \leq^\#, \vee^\#, \wedge^\# \rangle$ be a Galois connection.

- A function $f^\# : \mathcal{P}^\# \rightarrow \mathcal{P}^\#$ is a *sound approximation* of a function $f : \mathcal{P} \rightarrow \mathcal{P}$ if and only if $(f \circ \gamma)(y) \leq (\gamma \circ f^\#)(y)$ for all $y \in \mathcal{P}^\#$.
- If $\hat{f}^\#$ is a sound approximation of f such that for any function $f^\#$ that is a sound approximation of f we have $\hat{f}^\#(y) \leq f^\#(y)$ for all $y \in \mathcal{P}^\#$, then we call $\hat{f}^\#$ the *smallest sound approximation* of f .

Lemma 2.16: Constructing the Smallest Sound Approximation [28]

For a function $f : \mathcal{P} \rightarrow \mathcal{P}$ and a Galois connection

$$\langle \mathcal{P}, \leq, \vee, \wedge \rangle \xleftrightarrow[\alpha]{\gamma} \langle \mathcal{P}^\#, \leq^\#, \vee^\#, \wedge^\# \rangle$$

the smallest sound approximation $\hat{f}^\#$ of f is $\alpha \circ f \circ \gamma$.

Figure 2.1. Sound approximations of the function f . In (a), $f^\#$ is a sound approximation of f , so $(f \circ \gamma)(y) \leq (\gamma \circ f^\#)(y)$. In (b), as $\hat{f}^\# = \alpha \circ f \circ \gamma$, we have that $\hat{f}^\#$ is the smallest sound approximation of f .

Figure 2.1(a) shows a sound approximation $f^\#$ of a function f , while Figure 2.1(b) shows the smallest sound approximation of f , constructed using Lemma 2.16.

For a Galois connection formed via composition, the corresponding smallest sound abstraction of a function f can be expressed in terms of the smallest sound abstraction of f on the inner lattice of the composition (see Figure 2.2):

Lemma 2.17

Let

$$\mathcal{P} \xleftrightarrow[\alpha_1]{\gamma_1} \mathcal{P}^\# \xleftrightarrow[\alpha_2]{\gamma_2} \mathcal{P}^{\#\#}$$

be a Galois connection and $f: \mathcal{P} \rightarrow \mathcal{P}$ a function. Then the smallest sound abstraction $\hat{f}^{\#\#}$ of f on the lattice $\mathcal{P}^{\#\#}$ is $\alpha_2 \circ \hat{f}^\# \circ \gamma_2$, where $\hat{f}^\#$ is the smallest sound abstraction of f on the lattice $\mathcal{P}^\#$.

Proof. By Lemma 2.13(i), the abstraction function of $\mathcal{P} \xleftrightarrow[\alpha_1]{\gamma_1} \mathcal{P}^\# \xleftrightarrow[\alpha_2]{\gamma_2} \mathcal{P}^{\#\#}$ is $\alpha_2 \circ \alpha_1$, and the concretization function is $\gamma_1 \circ \gamma_2$. So by Lemma 2.16 the

Figure 2.2. Composition of the smallest sound approximations of the function f , as justified by Lemma 2.17. The smallest sound approximation $\hat{f}^{\#\#}$ of f on $\mathcal{P}^{\#\#}$ is equal to $\gamma_2 \circ \hat{f}^\# \circ \alpha_2$, where $\hat{f}^\#$ is the smallest sound approximation of f on $\mathcal{P}^\#$; hence, $\hat{f}^{\#\#} = \gamma_2 \circ \gamma_1 \circ f \circ \alpha_1 \circ \alpha_2$.

smallest sound approximation $\hat{f}^{\sharp\sharp}$ of f on $\mathcal{P}^{\sharp\sharp}$ is $\alpha_2 \circ \alpha_1 \circ f \circ \gamma_1 \circ \gamma_2$. But the smallest sound approximation $\hat{f}^\sharp$ of f on $\mathcal{P}^\sharp$ is $\alpha_1 \circ f \circ \gamma_1$, so we may rewrite $\hat{f}^{\sharp\sharp}$ as $\alpha_2 \circ \hat{f}^\sharp \circ \gamma_2$. $\square$

2.2 Propagator Functions

A *propagator function* maps a domain to a subset of itself. For a given constraint, a function that correctly distinguishes singleton domains on the basis of whether or not the sole assignment therein satisfies the constraint is said to *implement* the constraint:

Definition 2.18: Implementing a Constraint

A function p *implements* a constraint C if and only if

$$p(\{\langle a_1, \dots, a_n \rangle\}) = \{\langle a_1, \dots, a_n \rangle\}$$

for every assignment $\langle a_1, \dots, a_n \rangle \in C$.

A propagator function is a function on domains that meets the criteria in the following definition:

Definition 2.19: Propagator Function

A *propagator function* $p: \wp(\overline{\mathbf{D}}) \rightarrow \wp(\overline{\mathbf{D}})$, where $\overline{\mathbf{D}}$ is the set of candidate solutions, is a function from domains to domains that is, for every domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$:

- (i) *contracting*: $p(\mathcal{D}) \subseteq \mathcal{D}$
- (ii) *sound*: if $t \in \mathcal{D}$, then $p(\{t\}) \subseteq p(\mathcal{D})$

When a propagator function removes some tuples from a domain, we say that the propagator function *prunes* that domain.

Propagator contraction guarantees that propagator functions only remove values, and ensures termination of Algorithm 1.1. Propagator soundness implies that a propagator function that implements a constraint cannot prune any assignment that satisfies that constraint; propagator soundness is a weaker form of another commonly mentioned, but not mandatory, property of propagator functions, namely monotonicity:

Definition 2.20: Propagator Monotonicity

A propagator function p is *monotonic* if and only if

$$\mathcal{D} \subseteq \mathcal{D}' \implies p(\mathcal{D}) \subseteq p(\mathcal{D}')$$

for all $\mathcal{D}, \mathcal{D}' \subseteq \overline{\mathbf{D}}$.

Another interesting optional property of propagator functions is idempotency:

Definition 2.21: Propagator Idempotency

A propagator function p is *idempotent* if and only if $p(p(\mathcal{D})) = p(\mathcal{D})$ for all $\mathcal{D} \subseteq \overline{\mathbf{D}}$.

Two propagator functions that implement the same constraint may be compared in terms of their strength:

Definition 2.22: Relative Propagator Strength

The propagator function p_1 is *stronger* (respectively, *weaker*) than the propagator function p_2 , written $p_1 \leq p_2$ (respectively, $p_1 \geq p_2$), if and only if $p_1(\mathcal{D}) \subseteq p_2(\mathcal{D})$ (respectively, $p_1(\mathcal{D}) \supseteq p_2(\mathcal{D})$) for all domains $\mathcal{D} \subseteq \overline{\mathbf{D}}$. The propagator function p_1 is *strictly stronger* (respectively, *strictly weaker*) than the propagator function p_2 , written $p_1 \leqslant p_2$ (respectively, $p_1 \geqslant p_2$) if and only if $p_1 \leq p_2$ (respectively, $p_1 \geq p_2$) and there exists a domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ such that $p_1(\mathcal{D}) \subsetneq p_2(\mathcal{D})$ (respectively, $p_1(\mathcal{D}) \supsetneq p_2(\mathcal{D})$).

2.2.1 Consistency and Completeness

The absolute strength of a propagator function may be classified in two related ways, either by *consistency*, a property of the domain that results from propagation, or by *completeness*, a property of the propagator function itself. These two properties are now discussed in turn.

Consistency

Propagator strength is traditionally described by a property of domains called *consistency*, which is a set of necessary conditions on assignments for membership in the set of solutions [13].

In Section 1.2 domain consistency was introduced, under which every value of every variable domain participates in some assignment satisfying the constraint involving values in the variable domains of the other variables (see Example 1.2). Another example of a consistency is *bounds consistency*, under which every *bound* of every variable domain participates in some assignment satisfying the constraint involving values between the bounds of the variable domains of the other variables. Bounds consistency is implied by domain consistency, and a propagator function that maintains domain consistency for a given constraint is strictly stronger than a propagator function that maintains bounds consistency for the same constraint.

Example 2.1

For the constraint $\text{ALLDIFFERENT}(X_1, X_2, X_3, X_4)$, starting from the domain $\{1, 2\} \times \{1, 2\} \times \{2, 3, 4, 5\} \times \{4\}$, propagation to domain consistency leads to $\{1, 2\} \times \{1, 2\} \times \{3, 5\} \times \{4\}$, but propagation to bounds consistency leads to $\{1, 2\} \times \{1, 2\} \times \{3, 4, 5\} \times \{4\}$, since there do exist solutions to the constraint where X_3 takes its new lower bound 3 or its old (and new) upper bound 5, so that the infeasibility of the intermediate value 4 is not noticed.

Domain consistency and bounds consistency are simple enough to admit compact, understandable definitions.¹ More complicated consistencies, while equally interesting in theory, are significantly more difficult to describe in practice; Chapters 6 and 8 demonstrate the utility of the Galois-based framework developed in Part I for such complicated consistencies.

Completeness

The other measure of propagator strength is *completeness* [12, 63, 94], defined in terms of a *domain approximation*. A domain approximation is a subset of the set of candidate solutions that forms a complete lattice; in order to ensure the termination of Algorithm 1.1, a domain approximation must additionally be *unit approximate*, meaning that singleton domains are over-approximated by themselves [12]. An extensive function that over-approximates concrete domains by mapping them onto a domain approximation is a closure operation on the set of candidate solutions. Recall from Lemma 2.14 that for every closure operation ϕ on $\mathcal{P}$ there exists a Galois connection $\mathcal{P} \xrightleftharpoons[\alpha]{\gamma} \mathcal{P}^\#$ such that $\phi = \gamma \circ \alpha$. Intuitively, then, a propagator function is *complete* for a given closure operation ϕ if and only if the propagator function prunes at least as much as can be computed on the abstract lattice $\mathcal{P}^\#$ of the Galois connection corresponding to ϕ . Following the notation of Correia and Barahona [26], completeness is defined in relation to a closure operation thus:

Definition 2.23: ϕ -completeness [94]

Given n variables $\mathbf{X}$ with set $\bar{\mathbf{D}}$ of candidate solutions, a constraint C , and a closure operation $\phi: \bar{\mathbf{D}} \rightarrow \bar{\mathbf{D}}$, a propagator function p is *ϕ -complete* if and only if for any domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ we have $p(\mathcal{D}) \subseteq \phi(C \cap \phi(\mathcal{D}))$.

While ϕ -completeness only gives a lower bound on propagator strength, there is a unique, weakest ϕ -complete propagator function:

¹The history of bounds consistency demonstrates how much room there is for misunderstanding even relatively straightforward consistencies. Bounds consistency as defined in this section is often denoted $\text{bounds}(\mathbb{Z})$ consistency [87], to distinguish among the several consistencies that had previously shared the name bounds consistency (see, e.g., [13]). Variants of bounds consistency are discussed in Section 5.2.

Definition 2.24: ϕ -canonicity [94]

Given n variables $\mathbf{X}$ with set $\overline{\mathbf{D}}$ of candidate solutions, a constraint C , and a closure operation $\phi: \overline{\mathbf{D}} \rightarrow \overline{\mathbf{D}}$, the ϕ -canonical propagator function for C is a function p such that $p(\mathcal{D}) = C \cap \phi(C \cap \phi(\mathcal{D}))$.

Completeness and consistency are closely related properties, but further discussion is postponed until Chapter 3, at which point the relationship may be fully explicated within our Galois connection-based framework of propagation. Before presenting that framework, in the next section is defined the strongest possible consistency for constraint propagation: namely, *exact consistency*.

2.2.2 Exact Consistency

This section defines a consistency that corresponds to directly computing the set of solutions to a CSP. Naturally, this approach is impractical, as most interesting CSPs are, at best, NP-hard; in practice CP solvers reason not on the set of candidate solutions but rather on a more easily computed over-approximation. However, in the remainder of the thesis propagator functions that maintain this impractical consistency are referred to as a first step towards defining more practical consistencies and propagator functions.

Recall from Definition 1.1 in Section 1.2 that, for a CSP with variables $\mathbf{X}$ and set $\overline{\mathbf{D}}$ of candidate solutions, a domain, written $\mathcal{D}$, is a subset of $\overline{\mathbf{D}}$, i.e., a set of assignments. The set $\wp(\overline{\mathbf{D}})$ of all sets of assignments, when ordered by set inclusion, forms a complete lattice $\langle \wp(\overline{\mathbf{D}}), \subseteq, \cup, \cap \rangle$, which we call the *exact lattice*. The strongest consistency for any constraint is the consistency in which every assignment in a domain satisfies the constraint:

Definition 2.25: Exact Consistency

A domain $\mathcal{D} \in \wp(\overline{\mathbf{D}})$ is *exactly consistent* for a constraint C if and only if $\mathcal{D} \subseteq C$. We say that a set of assignments is *exactly consistent* for a CSP if and only if the set is exactly consistent for *every* constraint in the CSP.

Intuitively, exact consistency is the unique consistency that is a both necessary *and sufficient* condition on assignments for membership in the set of solutions. Thus, any set of assignments that is exactly consistent for a CSP corresponds exactly to a subset of the set of solutions for that CSP.

For any constraint a propagator function may be defined that keeps only those assignments that satisfy the constraint:

Definition 2.26: Exact Propagator Function

A constraint C has an associated *exact propagator function*:

$$C^E: \wp(\overline{\mathbf{D}}) \rightarrow \wp(\overline{\mathbf{D}}) \text{ where } C^E(\mathcal{T}) = C \cap \mathcal{T} \quad (2.1)$$

Example 2.2

Consider once more the CSP from Example 1.2 with variables $\langle X_1, X_2, X_3 \rangle$ and constraint $\text{ALLDIFFERENT}(X_1, X_2, X_3)$. By Definition 2.26, the exact propagator function for ALLDIFFERENT computes the intersection between the relation defining ALLDIFFERENT and a set of n -tuples:

$$\text{ALLDIFFERENT}^E(\mathcal{T}) = \{ \langle x_1, \dots, x_n \rangle \mid \forall i, j \in [1, n], i \neq j : x_i \neq x_j \} \cap \mathcal{T}$$

Applying this exact propagator function to the domain

$$\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}$$

yields:

$$\begin{aligned} & \text{ALLDIFFERENT}^E(\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}) \\ &= \{ \langle x_1, x_2, x_3 \rangle \mid x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3 \} \cap \\ & \quad \left\{ \begin{array}{l} \langle 1, 1, 1 \rangle, \langle 1, 1, 3 \rangle, \langle 1, 3, 1 \rangle, \langle 1, 3, 3 \rangle, \\ \langle 2, 1, 1 \rangle, \langle 2, 1, 3 \rangle, \langle 2, 3, 1 \rangle, \langle 2, 3, 3 \rangle, \\ \langle 3, 1, 1 \rangle, \langle 3, 1, 3 \rangle, \langle 3, 3, 1 \rangle, \langle 3, 3, 3 \rangle, \\ \langle 4, 1, 1 \rangle, \langle 4, 1, 3 \rangle, \langle 4, 3, 1 \rangle, \langle 4, 3, 3 \rangle \end{array} \right\} \\ &= \{ \langle 2, 1, 3 \rangle, \langle 2, 3, 1 \rangle, \langle 4, 1, 3 \rangle, \langle 4, 3, 1 \rangle \} \end{aligned}$$

The resulting set of tuples is comprised exactly of those assignments in the original domain that satisfy the constraint.

An exact propagator function is both idempotent and sound. In a somewhat contrived way it can be shown that an exact propagator function is also the ϕ -canonical propagator for the trivial Galois connection between the exact lattice and itself, where the closure operator is the identity function:

Proposition 2.27

Let C be a constraint on variables $\mathbf{X}$ with set $\overline{\mathbf{D}}$ of candidate solutions, with associated exact propagator function C^E , and let id be the identity function. Then C^E is both the id-canonical propagator function that implements C and the only id-complete propagator function that implements C .

Proof. That C^E is the id-canonical propagator function for C follows directly from Definition 2.23. We prove by contradiction that C^E is the only id-complete propagator function that implements C . Let C^{id} be an id-complete propagator function that implements C , such that $C^{\text{id}} \preceq C^E$. Then there is a domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ such that for some assignment $a \in \mathcal{D}$ we have $a \in C^E(\mathcal{D})$ and $a \notin C^{\text{id}}(\mathcal{D})$. By Definition 2.26 we have $a \in C$. So soundness implies that C^{id} does not implement C , which is a contradiction. $\square$

Because exact consistency is computationally impractical, CP solvers instead compute fixpoints based on some other, computationally practical consistency that is implied by exact consistency. Intuitively, where exact consistency is a set of necessary and sufficient conditions on assignments, the type of consistency sought is a set of necessary but *not* sufficient conditions on assignments. A propagator function that maintains this latter kind of consistency prunes from a domain some (but not necessarily all) assignments that do not satisfy the considered constraint. Starting from exact consistency, the next chapter demonstrates how to define a more relaxed consistency in terms of a Galois connection between the exact lattice and some abstraction thereof, and how to derive propagator functions that maintain that new consistency.

2.3 Related Work: Abstraction and Propagation

The similarity between constraint propagation and consistency on the one hand, and abstract interpretation on the other hand, was remarked as early as 1991 [20], in which it is observed that bounds reasoning for integer variables during constraint propagation is identical to interval abstraction in AI.² The connection is more developed in [5], where propagator fixpoint algorithms are described as chaotic iterations over partial orders, a characterization partly derived from AI. Abstract interpretation and domain consistency have also been studied in the context of constraint programming over floating-point variables, witness [9, 76] for instance, but this is beyond the scope of this dissertation, which focusses on *finite*-domain variables.

This thesis draws from two more recent lines of research ([46, 74, 98] and [12, 25, 26, 63, 94]). In both, consistency arises from the choice of an overapproximation of the variable domains. The formalisms defining these overapproximations differ between the two lines of research, yet are closely related to one another. At the same time, the two lines take quite different views as to the nature of propagators and, hence, the meaning of consistency.

²The focus of [20] is on *rules iteration*, another formalization of propagation, which is an alternative to consistency (see, e.g., [13]). Nevertheless, [20] includes a remark on the similarity between domain consistency and fixpoint calculation on an abstract lattice [27].

Figure 2.3. Three views of propagation for the constraint C . In (a), the propagator $C^\# = \alpha \circ C \circ \gamma$ is a function on the set $\wp(\mathcal{D})^\#$ of abstract domains. In (b), the propagator $C^{\phi\psi} = \psi \circ C \circ \phi$ is a function on the set $\wp(\mathcal{D})$ of concrete domains, encompassing the closure operations ϕ and ψ . In (c), we have $\phi = \gamma \circ \alpha = \psi$, and α and γ form a Galois connection between the set of concrete domains and the set of abstract domains. Then $C^\#$ is the smallest sound approximation of C on the set of abstract domains, and $C^{\phi\psi} = \gamma \circ \widehat{C^\#} \circ \alpha$.

2.3.1 Abstraction-based Frameworks

In the first line [46, 74, 98], which is based on AI, the over-approximation defining a consistency is given as a mapping of each concrete domain to an abstract domain. When both the set of concrete domains and the set of abstract domains form complete lattices, the connection between the two is a Galois connection, as shown for example by Gotlieb et al. [46] where domain consistency [62] and bounds($\mathbb{Z}$) [21, 101] consistency are characterized as computations on abstract domains that are related to concrete variable domains via a Galois connection (a view we revisit in Sections 2.2.2 and 4.2). For a set of abstract domains that is not a complete lattice, such as domains based on polyhedra [46] or octagons [74], there is no guarantee of the existence of a Galois connection, but Truchet et al. [98] provide a general definition of consistency as the greatest fixpoint of a computation on a set of abstract domains, irrespective of the presence of a Galois connection.

As shown in Figure 2.3(a), from this viewpoint a propagator that implements a constraint is a sound approximation of the constraint on the set of abstract domains, i.e., a contracting function on abstract domains (see, e.g., [74]). Consistency, then, is a limitation on the information available to a propagator: the choice of over-approximation determines the maximum amount of pruning that may be computed by a propagator. The choice of over-approximation is tied to the domain, and every propagator uses the same over-approximation. Mixing propagators of different consistencies is problematic, as the guarantee of contraction on the set of concrete domains is thereby lost. This view of consistency is therefore most appropriate when the choice of consistency is

representational, i.e., when the over-approximation is built into a CP solver’s implementation of the domain, and is shared by all propagators in the solver.

2.3.2 Closure-based Frameworks

In the second line of research [12, 25, 26, 63, 94], the over-approximation defining a consistency is defined in terms of a *domain approximation*, which is subset of the set of concrete domains that forms a complete lattice [12]. Consistency may then be expressed by means of the closure operator that maps a concrete domain to an elements of the domain approximation. As formulated by Tack [94] (and given a somewhat more general form by Correia and Barahona [25, 26]), the over-approximation defining a consistency is given as a pair of closure operators on the set of concrete domains. The idea, shown in Figure 2.3(b), is that a propagator maintains a particular consistency if and only if for every domain the propagator prunes at least as much as the composition of the following three operations: an over-approximation of the domain, followed by the pruning of all assignments that fail to satisfy the constraint, followed by a second over-approximation. In a sense, the first closure over-approximates the domain, while the second closure over-approximates the set of assignments that satisfy the constraint. Different combinations of closure operators yield consistencies corresponding to not only domain consistency or $\text{bounds}(\mathbb{Z})$ consistency, but also $\text{bounds}(\mathbb{R})$ consistency [21, 87], $\text{bounds}(\mathbb{D})$ consistency [21, 31], and range consistency [59].

Propagators in the closure-based approach are defined more conventionally than those in the abstraction-based approach; that is, in the closure-based approach propagators are seen as functions on the set of concrete domains, rather than functions on the set of abstract domains. Consistency, from the closure-based viewpoint, determines a minimum amount of pruning that a propagator maintaining that consistency must perform. The choice of over-approximations is tied to a propagator, determining what information the propagator ignores about the domain and the set of satisfying assignments. Since each propagator is contracting on the set of concrete domains, propagators maintaining different consistencies may be freely mixed. This view of consistency is most appropriate when the choice of consistency is *computational*, i.e., when performing less pruning for some constraints may result in a reduction of time complexity for the corresponding propagators.

2.4 A Sketch of a Galois-based Framework

Consistency in CP is a broad concept that encompasses the types of choices from both of these viewpoints. For example, bounds reasoning is a frequently employed over-approximation in CP, giving rise to *bounds consistency* for integer variables and *subset bounds consistency* for set variables. For integer

variables, bounds consistency is a computational choice. CP solvers do not typically represent integer variable domains as simple intervals. Propagators on these solvers are not limited to bounds reasoning when pruning domains; nevertheless, for some constraints it makes more sense to reason only on the bounds of variable domains, as the resulting propagators are computationally less expensive. In contrast, subset bounds consistency for set variables is a representational choice. CP solvers reduce space complexity by representing set variable domains by only their upper and lower bounds. There is therefore no reason for propagators for set constraints to calculate pruning beyond that justified by bounds reasoning.

The theoretical framework of Part I combines abstraction-based consistency and closure-based consistency. Central to this new framework is the following observation:

Remark 2.28

Any abstraction-based consistency in which there exists a Galois connection between the sets of concrete and abstract domains is equivalent to a closure-based consistency where the first and second closure operator are identical.

Remark 2.28 suggests that one may make use of features from both the abstraction-based and closure-based approaches. In the theoretical framework of Part I, the concept of consistency is defined in terms of abstract domains, thus every Galois connection between the set of concrete domains and the set of abstract domains induces a consistency. Propagators that maintain a particular consistency are derived via the corresponding closure operation, thus maintaining the conventional view of propagators as functions on concrete domains. The connection between consistency and propagator, illustrated in Figure 2.3(c), is the smallest sound abstraction of the implemented constraint: a contracting function on the set of abstract domains, well-defined due to the Galois connection, from which the weakest propagator implementing the constraint may be derived.

The unification of these two approaches yields several advantages. As shown in Chapter 3, defining a consistency by a Galois connection also yields the corresponding conditions on consistent domains. Galois connections are closed under the operations of composition and Cartesian product, so one may systematically construct complicated consistencies out of Galois connections arranged in series, in parallel, or both. One application for such a method is to construct consistencies for structured variable types, which can be particularly difficult to define clearly: see, e.g., the contrast between [IV], which defines an ad hoc consistency for bounded-length string variables, and Chapter 8, which systematically defines a similar consistency using Galois connections. Finally, smallest sound approximations are closed under the same operations as the

Galois connections they correspond to, so it is straightforward to derive propagators maintaining consistencies defined by constructed Galois connections.

To demonstrate possible applications of our framework, we identify the following three cases that call for complicated consistencies, which may be precisely described as Galois-based consistencies:

- Most commonly encountered consistencies are *uniform* with respect to variables, i.e., the current domain of every variable is over-approximated in the same way. However, for some constraints, it may be more computationally efficient to over-approximate some of the variables differently than others. As we describe in Section 6.2, the strength of such a propagator may be expressed as a *non-uniform* consistency.
- Consistency is also difficult to define for *structured* variable types (e.g., sets, tuples, graphs, or strings). As opposed to simple types like integer variables, which have totally ordered domains, the domains for structured types are often partially ordered, making consistencies for these types more complicated (see Section 5.2.1, where we discuss subset bounds consistency on set variables [14, 42]). As well, a structured variable type will often enforce a relationship among the various components of a single variable (e.g., between the length and the characters of a string variable, or between the vertices and edges of a graph variable). We discuss consistencies for structured variable types in Chapter 8.
- Complicated consistencies also arise when considering multiple variable types. CP solvers include several types of variables beyond integer variables, such as real number variables or set variables. Correctly describing consistencies for *mixed* variable types results in unwieldy consistency definitions, which can be simplified using Galois connections as described in Chapter 6.

3. Propagator Strength in the Abstract

The previous chapter introduced two closely related descriptions of propagation strength: consistency, a property of domains, and completeness, a property of propagators. This chapter elaborates the relationship between consistency and completeness using Galois connections, thus establishing a framework, based on Galois connections, connecting consistencies with the propagator functions that maintain them. This framework forms the central theoretical contribution of this thesis.

Section 3.1 shows how any Galois connection between the exact lattice and some abstraction thereof induces a new consistency that is a relaxation of exact consistency. As shown in Theorem 3.2, one may derive from such a Galois connection a condition that determines whether or not the induced consistency holds for a particular domain. Theorem 3.2 forms the bridge between these Galois-based consistencies and CP's more traditional consistency definitions. This relationship is developed further in Chapters 4 to 6.

Section 3.2 revisits the property of completeness, making explicit the connection between a Galois-based consistency and a propagator function that maintains it. For any Galois-based consistency, the corresponding completeness determines an upper bound on the strength of the propagator functions that maintains that consistency (see Theorem 3.5); furthermore, a Galois connection may be used to derive the unique weakest propagator function that maintains the induced consistency.

3.1 $\langle \alpha, \gamma \rangle$ -Consistency

Consistency may be generally defined in terms of a corresponding over-approximation of domains: a domain is consistent for a constraint C if and only if the smallest over-approximation of the domain is also the smallest over-approximation of some subset of C . In this section, we define consistency as a property arising from an over-approximation that is defined by a Galois connection. To those familiar with CP, this formulation of consistency may at first appear somewhat unusual. As we demonstrate at the end of this section, however, a Galois connection-based definition of a consistency may be systematically transformed into a more conventional consistency definition.

For variables $\mathbf{X}$ with set $\overline{\mathbf{D}}$ of candidate solutions and Galois connection $\wp(\overline{\mathbf{D}}) \xrightleftharpoons[\alpha]{\gamma} \wp(\overline{\mathbf{D}})^\#$ between the exact lattice and some abstract lattice, we call a domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ (i.e., a set of assignments) $\langle \alpha, \gamma \rangle$ -consistent for a constraint

if and only if the domain shares an abstract representation with some subset of the assignments that satisfy the constraint. This consistency is named parametrically with regard to the Galois connection; i.e., the consistency induced by a Galois connection with abstraction function τ and concretization function ρ is denoted as $\langle \tau, \rho \rangle$ -consistency. Formally:

Definition 3.1: $\langle \alpha, \gamma \rangle$ -consistency

Given a Galois connection $\wp(\bar{\mathbf{D}}) \xleftrightarrow[\alpha]{\gamma} \wp(\bar{\mathbf{D}})^\#$ between the exact lattice $\wp(\bar{\mathbf{D}})$ and some abstract lattice $\wp(\bar{\mathbf{D}})^\#$, we say that a domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ is $\langle \alpha, \gamma \rangle$ -consistent for a constraint C on variables $\mathbf{X}$ if and only if

$$\exists \mathcal{T} \subseteq C : \alpha(\mathcal{T}) = \alpha(\mathcal{D}) \quad (3.1)$$

Recall that for any Galois connection the upper and lower adjoints uniquely determine one another (Lemma 2.10); hence, (3.1) depends upon both α and γ , even though γ does not appear directly.

It is not immediately obvious from Definition 3.1 how to efficiently determine $\langle \alpha, \gamma \rangle$ -consistency for a particular domain. The following proposition gives an equivalent, but more directly computable, formulation:

Theorem 3.2: Determining $\langle \alpha, \gamma \rangle$ -consistency

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$. Let $C \in \mathbf{C}$ be a constraint, let

$$\langle \wp(\bar{\mathbf{D}}), \subseteq, \bigcup, \bigcap \rangle \xleftrightarrow[\alpha]{\gamma} \langle \wp(\bar{\mathbf{D}})^\#, \leq^\#, \bigvee^\#, \bigwedge^\# \rangle \quad (3.2)$$

be a Galois connection, and let C^E be an exact propagator function that implements C .

A domain $\mathcal{D} \in \bar{\mathbf{D}}$ is $\langle \alpha, \gamma \rangle$ -consistent for C if and only if

$$\alpha(\mathcal{D}) = (\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D}) \quad (3.3)$$

Proof. $\Leftarrow$: Let $\mathcal{D}$ be a domain satisfying (3.3), and let $\mathcal{T}$ be $(C^E \circ \gamma \circ \alpha)(\mathcal{D})$. By Definition 2.26 we have $\mathcal{T} = (\gamma \circ \alpha)(\mathcal{D}) \cap C \subseteq C$, and therefore (3.1) is satisfied.

$\Rightarrow$: Let $\mathcal{D}$ be a domain, and let $\mathcal{T}$ be a subset of C satisfying (3.1). Since $\wp(\bar{\mathbf{D}}) \xleftrightarrow[\alpha]{\gamma} \wp(\bar{\mathbf{D}})^\#$ is a Galois connection, we have $\mathcal{T} \subseteq (\gamma \circ \alpha)(\mathcal{T})$, and hence $\mathcal{T} \subseteq C \cap (\gamma \circ \alpha)(\mathcal{T})$. This implies that $\alpha(\mathcal{T}) \leq^\# \alpha((C \cap \gamma \circ \alpha)(\mathcal{T}))$. By Definition 2.26 we have

$$C \cap (\gamma \circ \alpha)(\mathcal{T}) = (C^E \circ \gamma \circ \alpha)(\mathcal{T})$$

and therefore

$$\alpha(\mathcal{D}) \leq^\# (\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D})$$

Figure 3.1. Consistency based on the Galois connection $\wp(\overline{\mathbf{D}}) \xleftrightarrow[\alpha]{\gamma} \wp(\overline{\mathbf{D}})^\#$. In (a), $\mathcal{D}$ is $\langle \alpha, \gamma \rangle$ -consistent, as it and its exactly propagated closure $(C^E \circ \gamma \circ \alpha)(\mathcal{D})$ have the same abstract representation. In (b), $\widehat{C^E}$ is equivalent to $\alpha \circ C^E \circ \gamma$ (shown with dotted lines), and therefore $\mathcal{D}' = (\gamma \circ \widehat{C^E} \circ \alpha)(\mathcal{D})$ is $\langle \alpha, \gamma \rangle$ -consistent.

Now, by Definition 2.26 we also have $(C^E \circ \gamma)(\alpha(\mathcal{T})) = C \cap \gamma(\alpha(\mathcal{T}))$, which must be a subset of $\gamma(\alpha(\mathcal{T}))$. By Lemma 2.9 it follows both that

$$\alpha((C^E \circ \gamma)(\alpha(\mathcal{T}))) \leq^\# \alpha(\gamma(\alpha(\mathcal{T})))$$

and that the latter is equal to $\alpha(\mathcal{T})$. Substituting $\alpha(\mathcal{D})$ for $\alpha(\mathcal{T})$ we get

$$(\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D}) \leq^\# \alpha(\mathcal{D})$$

As $\wp(\overline{\mathbf{D}})^\#$ is a poset, (3.3) follows directly from anti-symmetry. $\square$

Figure 3.1(a) illustrates the situation described by Theorem 3.2. The function $\gamma \circ \alpha$ calculates the α -induced closure of $\mathcal{D}$, or, in other words, the smallest element of the exact lattice that is both a superset of $\mathcal{D}$ and representable by an element of the abstract lattice $\wp(\overline{\mathbf{D}})^\#$. The exact propagator function C^E prunes all non-satisfying assignments from this closure of $\mathcal{D}$. If the resulting domain and the starting domain have the same abstract representation in $\wp(\overline{\mathbf{D}})^\#$, then the starting domain is $\langle \alpha, \gamma \rangle$ -consistent.

Theorem 3.2 may be reformulated as a set of conditions on domains that hews more closely to a conventional definition of consistency:

Corollary 3.3

Consider a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$. Let $C \in \mathbf{C}$ be a constraint, and let

$$\langle \wp(\overline{\mathbf{D}}), \subseteq, \bigcup, \bigcap \rangle \xleftrightarrow[\alpha]{\gamma} \langle \wp(\overline{\mathbf{D}})^\#, \leq^\#, \bigvee^\#, \bigwedge^\# \rangle$$

be a Galois connection.

A domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ is $\langle \alpha, \gamma \rangle$ -consistent for C if and only if $\alpha(\mathcal{D})$ is equal to $\alpha(\mathcal{D}')$ where $\mathcal{D}'$ is the set of every valid assignment $a \in \mathcal{D}$ such that $a \in C$, where an assignment a is *valid* if and only if $a \in (\gamma \circ \alpha)(\mathcal{D})$.

Proof. Let $\mathcal{D}'$ be the domain such that $\mathcal{D}' = C \cap (\gamma \circ \alpha)(\mathcal{D})$.

If we assume that $\alpha(\mathcal{D}) = \alpha(\mathcal{D}')$, then $\alpha(\mathcal{D})$ is equal to $\alpha(C \cap (\gamma \circ \alpha)(\mathcal{D}))$, which is equal to $(\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D})$; hence, by Theorem 3.2, $\mathcal{D}$ must be $\langle \alpha, \gamma \rangle$ -consistent. To prove the implication in the other direction, assume that $\mathcal{D}$ is $\langle \alpha, \gamma \rangle$ -consistent. By Theorem 3.2, we have that $(\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D})$ is equal to $\alpha(\mathcal{D})$, and therefore $\alpha(\mathcal{D}) = \alpha(\mathcal{D}')$. $\square$

Thus, in Chapters 4 to 6 where several conventional definitions of consistency are discussed, Theorem 3.2 will be used to demonstrate equivalence for corresponding definitions of $\langle \alpha, \gamma \rangle$ -consistency.

3.2 Propagator Completeness

Definition 3.1 and Theorem 3.2 define a consistency based on a Galois connection, but consistency is a property of domains, not of propagator functions. As our goal is to derive propagator functions that maintain a given $\langle \alpha, \gamma \rangle$ -consistency, we now revisit completeness, a related property of propagator functions that we first introduced in Section 2.2.1. We start with a discussion of smallest sound approximations of propagator functions in Section 3.2.1, which we then use in Section 3.2.2 to express $(\gamma \circ \alpha)$ -completeness, and in Section 3.2.3 to derive the weakest propagator function that maintains a given $\langle \alpha, \gamma \rangle$ -consistency, namely the $(\gamma \circ \alpha)$ -canonical propagator function.

3.2.1 Smallest Sound Approximations of Propagators

From Lemma 2.16 it follows that, for a Galois connection $\wp(\overline{\mathbf{D}}) \xleftrightarrow[\alpha]{\gamma} \wp(\overline{\mathbf{D}})^\sharp$, the smallest sound approximation on the abstract lattice $\wp(\overline{\mathbf{D}})^\sharp$ of an exact propagator function C^E for a constraint C is the function $\widehat{C}^\sharp: \wp(\overline{\mathbf{D}})^\sharp \rightarrow \wp(\overline{\mathbf{D}})^\sharp$ where $\widehat{C}^\sharp = \alpha \circ C^E \circ \gamma$. So the strongest pruning for constraint C that may be calculated on the abstract lattice $\wp(\overline{\mathbf{D}})^\sharp$ is given by the function $\gamma \circ \widehat{C}^\sharp \circ \alpha: \wp(\overline{\mathbf{D}}) \rightarrow \wp(\overline{\mathbf{D}})$: see Figure 3.1(b).

By Lemma 2.14, the function $\phi: \wp(\overline{\mathbf{D}}) \rightarrow \wp(\overline{\mathbf{D}})$ such that $\phi = \gamma \circ \alpha$ is a closure operation on domains. The function $\gamma \circ \widehat{C}^\sharp \circ \alpha$ can be rewritten as $\phi \circ C^E \circ \phi$, which helps to give an intuition of what the function does: the right ϕ over-approximates the domain, while the left ϕ over-approximates the set of assignments that satisfy the constraint. The function $\phi \circ C^E \circ \phi$ is, however, not a propagator function. As a composition of monotonic functions (i.e., closure operations and a set intersection) it is itself monotonic and therefore sound; however, it is not contracting, as demonstrated in the following example.

Example 3.1

Consider the interval closure operation on non-empty sets of integers, which we can write as $\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]}$ where $\alpha^{[\mathbb{Z}]}(\mathcal{S}) = [\min \mathcal{S}, \max \mathcal{S}]$ and $\gamma^{[\mathbb{Z}]}([\ell, u]) = \{s \in \mathbb{Z} \mid \ell \leq s \leq u\}$.

Let X_1 be a variable with domain $\text{dom}(X_1) = \{1, 2, 4\}$. Let $\text{ODD}(X_1)$ be a constraint that states “ X_1 is odd”, with associated exact propagator ODD^E . Let $\widehat{\text{ODD}}^{[\mathbb{Z}]}$ be the smallest sound abstraction of ODD^E on the lattice of interval closures, so that $\widehat{\text{ODD}}^{[\mathbb{Z}]} = \alpha^{[\mathbb{Z}]} \circ \text{ODD}^E \circ \gamma^{[\mathbb{Z}]}$. Then:

$$\begin{aligned} \left(\gamma^{[\mathbb{Z}]} \circ \widehat{\text{ODD}}^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\{1, 2, 4\}) &= \left(\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \circ \text{ODD}^E \circ \gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\{1, 2, 4\}) \\ &= \left(\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \circ \text{ODD}^E \right) (\{1, 2, 3, 4\}) \\ &= \left(\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\{1, 3\}) = \{1, 2, 3\} \not\subseteq \{1, 2, 4\} \end{aligned}$$

So $\gamma^{[\mathbb{Z}]} \circ \widehat{\text{ODD}}^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]}$ is not contracting.

Intuitively, the over-approximation of the set of assignments (i.e., the left closure operator) is contracting, but the over-approximation of the domain (i.e., the right closure operation) is not. If, however, the starting domain is closed under ϕ , then $\phi \circ C^E \circ \phi$ is guaranteed to be contracting:

Lemma 3.4

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$. Let $C \in \mathbf{C}$ be a constraint with associated exact propagator function C^E , and let ϕ be a closure operation on $\bar{\mathbf{D}}$. If a domain $\mathcal{D}$ is closed under ϕ , then $(\phi \circ C^E \circ \phi)(\mathcal{D})$ is a subset of $\mathcal{D}$.

Proof. Assume $\mathcal{D} = \phi(\mathcal{D})$. Then $(\phi \circ C^E \circ \phi)(\mathcal{D})$ is equal to $(\phi \circ C^E)(\mathcal{D})$, which by Definition 2.26 equals $\phi(C \cap \mathcal{D})$. By Definition 2.6(ii) ϕ is monotonic, so $\phi(C \cap \mathcal{D})$ is a subset of $\phi(C) \cap \phi(\mathcal{D}) \subseteq \phi(\mathcal{D})$. By assumption $\phi(\mathcal{D}) = \mathcal{D}$; therefore $\phi \circ C^E \circ \phi$ is contracting. $\square$

Example 3.2

Continuing from Example 3.1, let $\text{dom}(X_1) = \{1, 2, 3, 4\}$. Since $\text{dom}(X_1)$ is already closed under $\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]}$, from Lemma 3.4 it is expected that:

$$\left(\gamma^{[\mathbb{Z}]} \circ \widehat{\text{ODD}}^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\text{dom}(X_1)) \subseteq \text{dom}(X_1)$$

Indeed, this is the case:

$$\left(\gamma^{[\mathbb{Z}]} \circ \widehat{\text{ODD}}^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\{1, 2, 3, 4\})$$

$$\begin{aligned}
&= (\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \circ \text{ODD}^E \circ \gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]}) (\{1, 2, 3, 4\}) \\
&= (\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \circ \text{ODD}^E) (\{1, 2, 3, 4\}) \\
&= (\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]}) (\{1, 3\}) = \{1, 2, 3\} \subseteq \{1, 2, 3, 4\}
\end{aligned}$$

3.2.2 $(\gamma \circ \alpha)$ -Completeness

While $\gamma \circ \hat{C}^\# \circ \alpha$ is not a propagator function, it is nonetheless interesting. For a constraint C with associated exact propagator function C^E , any propagator function $C^\#$ that implements C is ϕ -complete for $\phi = \gamma \circ \alpha$ if and only if $C^\#(\mathcal{D})$ is a subset of $(\gamma \circ \hat{C}^\# \circ \alpha)(\mathcal{D})$. Thus, there exists a close relationship between ϕ -completeness and the smallest sound approximation of an exact propagator function, and hence between ϕ -completeness and $\langle \alpha, \gamma \rangle$ -consistency. In fact, any $(\gamma \circ \alpha)$ -complete propagator function maintains $\langle \alpha, \gamma \rangle$ -consistency:

Theorem 3.5

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$. Let $C \in \mathbf{C}$ be a constraint that is implemented by the exact propagator function C^E , let

$$\langle \wp(\bar{\mathbf{D}}), \subseteq, \bigcup, \bigcap \rangle \xleftrightarrow[\alpha]{\gamma} \langle \wp(\bar{\mathbf{D}})^\#, \leq^\#, \bigvee^\#, \bigwedge^\# \rangle$$

be a Galois connection, and let $C^\#$ be a $(\gamma \circ \alpha)$ -complete propagator function that implements C .

If a domain $\mathcal{D} \in \bar{\mathbf{D}}$ is a fixpoint of $C^\#$, then $\mathcal{D}$ is $\langle \alpha, \gamma \rangle$ -consistent.

Proof. Assume $\mathcal{D}$ is a fixpoint of $C^\#$, so $C^\#(\mathcal{D}) = \mathcal{D}$.

We wish to demonstrate that $\alpha(\mathcal{D}) = (\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D})$. Since $C^\#$ is a $(\gamma \circ \alpha)$ -complete propagator function, we have:

$$\begin{aligned}
C^\#(\mathcal{D}) &\subseteq (\gamma \circ \alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D}) && [\text{by Def. 2.23}] \\
\mathcal{D} &\subseteq (\gamma \circ \alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D}) && [\text{by assumption}] \\
\alpha(\mathcal{D}) &\leq^\# (\alpha \circ \gamma \circ \alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D}) && [\text{by Lemma 2.9(ii)}] \\
\alpha(\mathcal{D}) &\leq^\# (\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D}) && [\text{by Lem. 2.9(iii)}]
\end{aligned}$$

Furthermore, since $C^\#$ is a propagator, we have:

$$\begin{aligned}
C^\#((\gamma \circ \alpha)(\mathcal{D})) &\subseteq (\gamma \circ \alpha)(\mathcal{D}) && [\text{by Def. 2.19(i)}] \\
(\alpha \circ C^\# \circ \gamma \circ \alpha)(\mathcal{D}) &\leq^\# (\alpha \circ \gamma \circ \alpha)(\mathcal{D}) && [\text{by Lem. 2.9(ii)}] \\
(\alpha \circ C^\# \circ \gamma \circ \alpha)(\mathcal{D}) &\leq^\# \alpha(\mathcal{D}) && [\text{by Lem. 2.9(ii)}]
\end{aligned}$$

Since $\wp(\overline{\mathbf{D}})^\sharp$ is a poset, it follows from anti-symmetry that $\alpha(\mathcal{D})$ is equal to $(\alpha \circ C^E \circ \gamma \circ \alpha)(\mathcal{D})$, so by Theorem 3.2 $\mathcal{D}$ is $\langle \alpha, \gamma \rangle$ -consistent. $\square$

3.2.3 $(\gamma \circ \alpha)$ -Canonicity

Recall from Definition 2.24 in Section 2.2.1 that for every constraint there exists a unique, weakest ϕ -complete propagator function, called the ϕ -canonical propagator function. Note that ϕ -canonical propagator functions (Definition 2.24) can also be rewritten in terms of an exact propagator function: a propagator function C^ϕ that implements a constraint C with associated exact propagator function C^E is ϕ -canonical if and only if $C^\phi(\mathcal{D})$ is equal to $\mathcal{D} \cap (\phi \circ C^E \circ \phi)(\mathcal{D})$, for any current domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$. A ϕ -canonical propagator function is clearly contracting, and monotonic (and therefore sound) as it is a composition of set intersection and closure operations. In contrast to the function $\gamma \circ \widehat{C}^\sharp \circ \alpha$, a ϕ -canonical propagator function is not idempotent in the general case, as we demonstrate in Section 3.2.4.

For any Galois connection relating the exact lattice to an abstract lattice, the corresponding $(\gamma \circ \alpha)$ -canonical propagator function maintains $\langle \alpha, \gamma \rangle$ -consistency, and may be derived using the exact propagator function and the concretization and abstraction functions. In Chapters 4 to 6 and 8, we derive $(\gamma \circ \alpha)$ -canonical propagator functions for several consistencies, while in Section 9.1 we demonstrate the use of derived $(\gamma \circ \alpha)$ -canonical propagator functions in the automatic generation of solver-specific propagator algorithms. Note that this derivation is wholly mechanical and could, in principle, be automated by, e.g., SPECWARE, a category-theoretic, general-purpose synthesis tool [92].

Before proceeding, though, we digress briefly to discuss the conditions under which a ϕ -canonical propagator function is idempotent.

3.2.4 ϕ -Canonicity and Idempotency

A ϕ -canonical propagator function is not generally idempotent, as demonstrated by the following example.

Example 3.3

Returning to Example 3.1, we denote as $\text{ODD}^{[\mathbb{Z}]}$ the $(\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]})$ -canonical propagator function that implements the constraint ODD . So by Definition 2.24 we have:

$$\begin{aligned} \text{ODD}^{[\mathbb{Z}]}(\{1, 2, 4\}) &= \{1, 2, 4\} \cap \left(\gamma^{[\mathbb{Z}]} \circ \widehat{\text{ODD}^{[\mathbb{Z}]}} \circ \alpha^{[\mathbb{Z}]} \right) (\{1, 2, 4\}) \\ &= \{1, 2, 4\} \cap \{1, 2, 3\} = \{1, 2\} \end{aligned}$$

Figure 3.2. Illustration of idempotency conditions for $(\gamma \circ \alpha)$ -canonical propagator functions. $\widehat{\text{ODD}}^{[Z]}$ is the smallest sound approximation of ODD^E on the abstract lattice $[Z]$, and the $(\gamma \circ \alpha)$ -canonical implementing ODD is $\text{ODD}^{[Z]}$, which is equal to $\gamma^{[Z]} \circ \widehat{\text{ODD}}^{[Z]} \circ \alpha^{[Z]}$. In (b), where domain $\{1, 2, 3, 4\}$ is ϕ -closed, $\text{ODD}^{[Z]}$ calculates a fixpoint; in (a), where domain $\{1, 2, 4\}$ is not ϕ -closed, it does not.

So $\text{ODD}^{[Z]}$ has pruned only one of the two even values from $\{1, 2, 4\}$. The problem is the initial closure operation $(\gamma^{[Z]} \circ \alpha^{[Z]})(\{1, 2, 4\})$, shown in Figure 3.2(a), which over-approximates the current domain by adding the value 3. As 3 satisfies the constraint, it is not pruned until the intersection with $\{1, 2, 4\}$, which occurs last. A second iteration of $\text{ODD}^{[Z]}$ prunes the value 2 from the domain, and reaches fixpoint:

$$\begin{aligned} \text{ODD}^{[Z]}(\{1, 2\}) \\ = \{1, 2\} \cap \left(\gamma^{[Z]} \circ \widehat{\text{ODD}}^{[Z]} \circ \alpha^{[Z]} \right) (\{1, 2\}) = \{1, 2\} \cap \{1\} = \{1\} \end{aligned}$$

The resulting current domain $\text{dom}(X_1) = \{1\}$ is $\langle \alpha^{[Z]}, \gamma^{[Z]} \rangle$ -consistent.

A ϕ -canonical propagator is, however, idempotent for an interesting subset of domains, namely those that are closed under ϕ , i.e., the same set of domains for which $\gamma \circ \widehat{C}^\# \circ \alpha$ is contracting, as shown in Lemma 3.4. To prove this, we first show that $\gamma \circ \widehat{C}^\# \circ \alpha$ is idempotent for all domains:

Lemma 3.6

Consider a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$. Let $C \in \mathbf{C}$ be a constraint that is implemented by the exact propagator function C^E .

If ϕ is a closure operation on $\overline{\mathbf{D}}$, then $\phi \circ C^E \circ \phi$ is idempotent.

Proof. Assume ϕ is a closure operation on $\overline{\mathbf{D}}$, and let domain $\mathcal{D}$ be a subset of $\overline{\mathbf{D}}$. We begin by considering the elements of the ϕ -closure of $\mathcal{D}$ that satisfy the

constraint C , i.e., $C \cap \phi(\mathcal{D})$:

$$\begin{aligned}
C \cap \phi(\mathcal{D}) &\subseteq \phi(C \cap \phi(\mathcal{D})) && [\text{by Def. 2.6(i)}] \\
C \cap C \cap \phi(\mathcal{D}) &\subseteq C \cap \phi(C \cap \phi(\mathcal{D})) \\
C \cap \phi(\mathcal{D}) &\subseteq C \cap \phi(C \cap \phi(\mathcal{D})) \\
\phi(C \cap \phi(\mathcal{D})) &\subseteq \phi(C \cap \phi(C \cap \phi(\mathcal{D}))) && [\text{by Def. 2.6(ii)}] \\
\phi(C \cap \phi(\mathcal{D})) &\subseteq \phi(C \cap \phi(\phi(C \cap \phi(\mathcal{D})))) && [\text{by Def. 2.6(iii)}]
\end{aligned}$$

and therefore

$$(\phi \circ C^E \circ \phi)(\mathcal{D}) \subseteq (\phi \circ C^E \circ \phi)((\phi \circ C^E \circ \phi)(\mathcal{D})) \quad (3.4)$$

The reverse of this inclusion holds as well:

$$\begin{aligned}
&(\phi \circ C^E \circ \phi)((\phi \circ C^E \circ \phi)(\mathcal{D})) \\
&= \phi(C \cap \phi(\phi(C \cap \phi(\mathcal{D})))) \\
&= \phi(C \cap \phi(C \cap \phi(\mathcal{D}))) && [\text{by Def. 2.6(iii)}] \\
&\subseteq \phi(C) \cap \phi(\phi(C \cap \phi(\mathcal{D}))) && [\text{by Def. 2.6(ii)}] \\
&\subseteq \phi(\phi(C \cap \phi(\mathcal{D}))) \\
&= \phi(C \cap \phi(\mathcal{D})) && [\text{by Def. 2.6(iii)}]
\end{aligned}$$

Therefore $(\phi \circ C^E \circ \phi)((\phi \circ C^E \circ \phi)(\mathcal{D}))$ is equal to $(\phi \circ C^E \circ \phi)(\mathcal{D})$, and $\phi \circ C^E \circ \phi$ is idempotent. $\square$

We now demonstrate that, for ϕ -closed domains, a ϕ -canonical propagator is idempotent:

Proposition 3.7

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$. Let $C \in \mathbf{C}$ be a constraint that is implemented by the exact propagator function C^E , let ϕ be a closure operation on $\bar{\mathbf{D}}$, and let C^ϕ be the ϕ -canonical propagator for C . If a domain $\mathcal{D}$ is closed under ϕ , then $C^\phi(C^\phi(\mathcal{D})) = C^\phi(\mathcal{D})$.

Proof. Assume $\phi(\mathcal{D}) = \mathcal{D}$. By Lemma 3.4 $(\phi \circ C^E \circ \phi)(\mathcal{D})$ is a subset of $\mathcal{D}$. It follows that:

$$(\phi \circ C^E \circ \phi)(\mathcal{D}) = \mathcal{D} \cap (\phi \circ C^E \circ \phi)(\mathcal{D}) = C^\phi(\mathcal{D}) \quad (3.5)$$

So by Lemma 3.6, $C^\phi(C^\phi(\mathcal{D})) = C^\phi(\mathcal{D})$. $\square$

The following exemplifies the situation described in Proposition 3.7:

Example 3.4

Returning to Example 3.3, consider the pruning of the domain $\text{dom}(X_1) = \{1, 2, 3, 4\}$ by the $(\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]})$ -canonical propagator function $\text{ODD}^{[\mathbb{Z}]}$:

$$\begin{aligned} \text{ODD}^{[\mathbb{Z}]}(\{1, 2, 3, 4\}) &= \{1, 2, 3, 4\} \cap \left(\gamma^{[\mathbb{Z}]} \circ \widehat{\text{ODD}}^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\{1, 2, 3, 4\}) \\ &= \{1, 2, 3, 4\} \cap \{1, 2, 3\} = \{1, 2, 3\} \end{aligned}$$

The domain $\text{dom}(X_1)$ is closed under the interval closure operation, as we have:

$$\left(\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\text{dom}(X_1)) = \gamma^{[\mathbb{Z}]} ([1, 4]) = \{1, 2, 3, 4\}$$

Hence, Proposition 3.7 predicts that a second iteration of $\text{ODD}^{[\mathbb{Z}]}$ will yield no additional pruning. This is in fact the case, as we have:

$$\begin{aligned} \text{ODD}^{[\mathbb{Z}]}(\{1, 2, 3\}) &= \{1, 2, 3\} \cap \left(\gamma^{[\mathbb{Z}]} \circ \widehat{\text{ODD}}^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]} \right) (\{1, 2, 3\}) \\ &= \{1, 2, 3\} \cap \{1, 2, 3\} = \{1, 2, 3\} \end{aligned}$$

So $\{1, 2, 3\}$ is a fixpoint of $\text{ODD}^{[\mathbb{Z}]}$ (see Figure 3.2(b)).

* * *

Proposition 3.7 is significant because of the following case: if the abstract lattice of the ϕ -induced Galois connection defines the domain representation used in a CP solver, then only ϕ -closed domains are encountered. By Proposition 3.7, a ϕ -canonical propagator function will always be idempotent in that case, as only ϕ -closed domains can be represented in the CP solver. One such example may be found in Chapter 4, which covers a consistency informally introduced in Section 1.2, namely *domain consistency*, which is based on the Cartesian closure operation. Additional examples may be found in Chapter 8, which will introduce a consistency for a bounded-length string variable implementation.

4. Cartesian Consistencies

As discussed in Chapter 1, treating the domain of each variable separately during propagation is the fundamental abstraction in most CP solvers. Section 4.1 formally defines this *Cartesian approximation* of domains. In Section 4.2, the strongest consistency based on the Cartesian approximation, called *domain consistency*, is defined as an $\langle \alpha, \gamma \rangle$ -consistency. Finally, a general framework for defining weaker Cartesian consistencies is presented in Section 4.3. Of particular note is Theorem 4.4, which shows how to define conditions on domains maintaining such a $\langle \alpha, \gamma \rangle$ -consistency in terms of conditions on the individual variables involved.

4.1 Cartesian Approximation

For sets $\mathcal{A}_1, \dots, \mathcal{A}_n$, the *Cartesian closure* of a set $\mathcal{T} \subseteq \mathcal{A}_1 \times \dots \times \mathcal{A}_n$ of tuples is the smallest Cartesian product $\mathcal{T}_1 \times \dots \times \mathcal{T}_n$ such that $\mathcal{T}_i$ is the set of all values occurring at the i -th index of some tuple $t \in \mathcal{T}$ for all $i \in [1, n]$, and the *Cartesian abstraction* of $\mathcal{T}$ is the tuple $\langle \mathcal{T}_1, \dots, \mathcal{T}_n \rangle$ of sets. For example, for the set of tuples $\{\langle 1, 2, 1 \rangle, \langle 1, 2, 2 \rangle, \langle 2, 1, 1 \rangle\}$ the Cartesian closure is the set of tuples $\{1, 2\} \times \{1, 2\} \times \{1, 2\}$, and the Cartesian abstraction is the tuple of sets $\langle \{1, 2\}, \{1, 2\}, \{1, 2\} \rangle$. We call the function mapping a set of tuples to its Cartesian closure the *Cartesian closure operator*. The Cartesian closure operator is unit approximate, as it maps any singleton set of tuples $\mathcal{T} = \{\langle t_1, \dots, t_n \rangle\}$ to the set of tuples $\{t_1\} \times \dots \times \{t_n\}$, which is equal to $\mathcal{T}$.

The set of Cartesian abstractions of the subsets of $\mathcal{A}_1 \times \dots \times \mathcal{A}_n$ is $\wp(\mathcal{A}_1) \times \dots \times \wp(\mathcal{A}_n)$, which, as the Cartesian product of n powerset lattices, is itself a complete lattice under the component-wise set inclusion order $\subseteq$ (i.e., for $x, y \in \mathcal{A}_1 \times \dots \times \mathcal{A}_n$, we have $x \subseteq y$ means $x_i \subseteq y_i$ for all $i \in [1, n]$), with lub $\cup$ (component-wise union) and glb $\cap$ (component-wise intersection) defined similarly. Thus, by Lemma 2.14, there must exist a Galois connection such that $\gamma^D \circ \alpha^D$ is the Cartesian closure operator:

$$\wp(\mathcal{A}_1 \times \dots \times \mathcal{A}_n) \xrightleftharpoons[\alpha^D]{\gamma^D} (\wp(\mathcal{A}_1) \times \dots \times \wp(\mathcal{A}_n)) \quad (4.1)$$

We call the Galois connection defined in (4.1) the *Cartesian approximation*.

We call the function α^D from sets of tuples to their Cartesian abstractions the *Cartesian abstraction function*, and define it as follows:

$$\alpha^D(\mathcal{T}) = \left\langle \begin{array}{c} \{v_1 \in \mathcal{A}_1 \mid \langle v_1, \dots, v_n \rangle \in \mathcal{T}\}, \\ \dots, \\ \{v_n \in \mathcal{A}_n \mid \langle v_1, \dots, v_n \rangle \in \mathcal{T}\} \end{array} \right\rangle \quad (4.2)$$

We now define the upper adjoint of α^D , the function γ^D from Cartesian abstractions to Cartesian closures, such that $\gamma^D \circ \alpha^D$ is the Cartesian closure operator. We call γ^D the *Cartesian concretization function*, and define it as follows:

$$\gamma^D(\mathcal{V}_1, \dots, \mathcal{V}_n) = \mathcal{V}_1 \times \dots \times \mathcal{V}_n \quad (4.3)$$

Recall from Definition 1.1 that for a domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$, the domain of a variable X_i , written $\text{dom}(X_i)$, is the projection of $\mathcal{D}$ onto the set of values at the i -th index. Therefore, when the Cartesian abstraction function is applied to a domain $\mathcal{D}$ for the variables $\mathbf{X}$, we have:

$$\alpha^D(\mathcal{D}) = \langle \text{dom}(X_1), \dots, \text{dom}(X_n) \rangle \quad (4.4)$$

4.2 Domain Consistency

We now revisit a consistency previously introduced in Section 1.2: domain consistency [62], which requires that every value in the domain $\text{dom}(X_i) \subseteq \overline{\mathcal{D}}_i$ of each variable X_i participates in some assignment satisfying the constraint involving values in the domains of the other variables. Formally:

Definition 4.1: Domain consistency

Consider a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\overline{\mathbf{D}} = \overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n$ of candidate solutions. Let $C \in \mathbf{C}$ be a constraint that is implemented by the exact propagator function C^E .

A domain $\mathcal{D}$ is *domain consistent* for C if and only if for each variable X_i such that $1 \leq i \leq n$, and for each value $d \in \text{dom}(X_i)$, there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ such that $d_i = d$, where an assignment $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if $d_\ell \in \text{dom}(X_\ell)$ for all $1 \leq \ell \leq n$.

The lattice $\langle \wp(\overline{\mathcal{D}}_1) \times \dots \times \wp(\overline{\mathcal{D}}_n), \subseteq, \cup, \cap \rangle$ of Cartesian abstractions of domains is called the *domain lattice*. The Cartesian abstraction function (4.2) and the Cartesian concretization function (4.3) form the following Galois connection between the exact lattice and the domain lattice:

$$\left\langle \wp(\wp(\overline{\mathcal{D}}_1) \times \dots \times \wp(\overline{\mathcal{D}}_n)), \subseteq, \cup, \cap \right\rangle \xleftrightarrow[\alpha^D]{\gamma^D} \left\langle \wp(\overline{\mathcal{D}}_1) \times \dots \times \wp(\overline{\mathcal{D}}_n), \subseteq, \cup, \cap \right\rangle \quad (4.5)$$

We now demonstrate that the consistency induced by this Galois connection is domain consistency.

Proposition 4.2

$\langle \alpha^D, \gamma^D \rangle$ -consistency is equivalent to domain consistency.

Proof. As $\gamma^D \circ \alpha^D$ is the Cartesian closure operation, we can rewrite Definition 4.1 as $\alpha^D(\mathcal{D}) = (\alpha^D \circ C^E \circ \gamma^D \circ \alpha^D)(\mathcal{D})$; hence, by Theorem 3.2, domain consistency and $\langle \alpha^D, \gamma^D \rangle$ -consistency are equivalent. $\square$

By Theorem 3.5, any $(\gamma^D \circ \alpha^D)$ -complete propagator function implementing a constraint C maintains $\langle \alpha^D, \gamma^D \rangle$ -consistency for C . The weakest such propagator function is the $(\gamma^D \circ \alpha^D)$ -canonical propagator function C^D which is the function defined by:

$$C^D(\mathcal{D}) = \mathcal{D} \cap (\gamma^D \circ \widehat{C}^D \circ \alpha^D)(\mathcal{D}) \quad (4.6)$$

where $\widehat{C}^D = \alpha^D \circ C^E \circ \gamma^D$ is the smallest sound abstraction of the exact propagator function C^E on the domain lattice.

Together, Proposition 4.2 and Theorem 3.5 show that for any constraint C with exact propagator function C^E , the $(\gamma^D \circ \alpha^D)$ -canonical propagator function is the weakest propagator function that maintains domain consistency.

Example 4.1

Let ALLDIFFERENT^E be the exact propagator function for the constraint ALLDIFFERENT , as defined in Example 2.2. By Lemma 2.16, the smallest sound approximation of ALLDIFFERENT^E on the domain lattice, the function ALLDIFFERENT^D , is defined by:

$$\begin{aligned} \widehat{\text{ALLDIFFERENT}}^D(\mathcal{V}_1, \dots, \mathcal{V}_n) &= \\ &= (\alpha^D \circ \text{ALLDIFFERENT}^E \circ \gamma^D)(\mathcal{V}_1, \dots, \mathcal{V}_n) \\ &= (\alpha^D \circ \text{ALLDIFFERENT}^E)(\mathcal{V}_1 \times \dots \times \mathcal{V}_n) \\ &= \alpha^D(\{ \langle v_1, \dots, v_n \rangle \in \mathcal{V}_1 \times \dots \times \mathcal{V}_n \mid \forall i, j \in [1, n], i \neq j : v_i \neq v_j \}) \\ &= \left\langle \begin{array}{l} \left\{ v_1 \in \mathcal{V}_1 \mid \exists \langle v_1, \dots, v_n \rangle \in \{v_1\} \times \mathcal{V}_2 \times \dots \times \mathcal{V}_n : \right. \\ \quad \left. \forall i, j \in [1, n], i \neq j : v_i \neq v_j \right\} \\ \dots, \\ \left\{ v_n \in \mathcal{V}_n \mid \exists \langle v_1, \dots, v_n \rangle \in \mathcal{V}_1 \times \dots \times \mathcal{V}_{n-1} \times \{v_n\} : \right. \\ \quad \left. \forall i, j \in [1, n], i \neq j : v_i \neq v_j \right\} \end{array} \right\rangle \end{aligned}$$

The n -ary $(\gamma^D \circ \alpha^D)$ -canonical propagator function for ALLDIFFERENT is derived as follows:

$$\begin{aligned} \text{ALLDIFFERENT}^D(\mathcal{D}) &= \\ &= \mathcal{D} \cap (\gamma^D \circ \widehat{\text{ALLDIFFERENT}}^D \circ \alpha^D)(\mathcal{D}) \\ &= \mathcal{D} \cap (\gamma^D \circ \widehat{\text{ALLDIFFERENT}}^D)(\text{dom}(X_1), \dots, \text{dom}(X_n)) \end{aligned}$$

$$\begin{aligned}
&= \mathcal{D} \cap \gamma^D \left(\left\{ \left\{ \begin{array}{l} x_1 \in \text{dom}(X_1) \\ \text{dom}(X_1) \end{array} \middle| \begin{array}{l} \exists \langle x_1, \dots, x_n \rangle \in \\ \{x_1\} \times \text{dom}(X_2) \times \dots \times \text{dom}(X_n) : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\}, \dots, \right. \\
&\quad \left. \left\{ \begin{array}{l} x_n \in \text{dom}(X_n) \\ \text{dom}(X_n) \end{array} \middle| \begin{array}{l} \exists \langle x_1, \dots, x_n \rangle \in \\ \text{dom}(X_1) \times \text{dom}(X_2) \times \dots \times \{x_n\} : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right) \\
&= \mathcal{D} \cap \left(\left\{ \begin{array}{l} x_1 \in \text{dom}(X_1) \\ \text{dom}(X_1) \end{array} \middle| \begin{array}{l} \exists \langle x_1, \dots, x_n \rangle \in \\ \{x_1\} \times \text{dom}(X_2) \times \dots \times \text{dom}(X_n) : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right. \\
&\quad \times \dots \times \\
&\quad \left. \left\{ \begin{array}{l} x_n \in \text{dom}(X_n) \\ \text{dom}(X_n) \end{array} \middle| \begin{array}{l} \exists \langle x_1, \dots, x_n \rangle \in \\ \text{dom}(X_1) \times \text{dom}(X_2) \times \dots \times \{x_n\} : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right)
\end{aligned}$$

Applying ALLDIFFERENT^D to the starting domain from Example 1.2, namely $\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}$, we get:

$$\begin{aligned}
&\text{ALLDIFFERENT}^D(\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}) \\
&= \{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\} \cap \\
&\quad \left\{ \begin{array}{l} x \in \{1, 2, 3, 4\} \\ \{1, 2, 3, 4\} \end{array} \middle| \begin{array}{l} \exists y \in \{1, 3\}, z \in \{1, 3\} : \\ x \neq y \wedge x \neq z \wedge y \neq z \end{array} \right\} \times \\
&\quad \left\{ \begin{array}{l} y \in \{1, 3\} \\ \{1, 3\} \end{array} \middle| \begin{array}{l} \exists x \in \{1, 2, 3, 4\}, z \in \{1, 3\} : \\ x \neq y \wedge x \neq z \wedge y \neq z \end{array} \right\} \times \\
&\quad \left\{ \begin{array}{l} z \in \{1, 3\} \\ \{1, 3\} \end{array} \middle| \begin{array}{l} \exists x \in \{1, 2, 3, 4\}, y \in \{1, 3\} : \\ x \neq y \wedge x \neq z \wedge y \neq z \end{array} \right\} \\
&= \{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\} \cap \{2, 4\} \times \{1, 3\} \times \{1, 3\} \quad (4.7) \\
&= \{2, 4\} \times \{1, 3\} \times \{1, 3\}
\end{aligned}$$

Note that $\text{ALLDIFFERENT}^E(\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\})$ is a subset of $\text{ALLDIFFERENT}^D(\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\})$. For example, $\langle 2, 1, 1 \rangle$ is an element of the latter, but not the former (given in Example 2.2).

Also, as suggested by Lemma 3.4, the set intersection calculated at (4.7) was not required in this example, as the starting domain $\{1, 2, 3, 4\} \times \{1, 3\} \times \{1, 3\}$ is closed under the Cartesian closure operator; by Proposition 3.7, the resulting domain must be a fixpoint of ALLDIFFERENT^D . If the starting domain is not closed, then this set intersection operation is required to ensure contraction, and the resulting intersection is not guaranteed to be a fixpoint (see Example 3.3 above).

4.3 Relaxations of Domain Consistency

For some constraints, propagator functions that maintain domain consistency do not have known, efficient propagator algorithms, while weaker propagator functions do have efficient propagator algorithms. One way to define a consistency that such a propagator function might maintain is by composition of the Galois connection defining domain consistency (4.5) with a second Galois connection. As the domain lattice is a Cartesian product of sets, this second Galois connection may be constructed as the Cartesian product of Galois connections, each on the domain of one variable. The resulting Galois connection is called a *Cartesian domain abstraction*:

Definition 4.3: Cartesian Domain Abstraction

For an initial domain $\overline{\mathcal{D}}_1 \times \cdots \times \overline{\mathcal{D}}_n$, a Galois connection of the following form is called a *Cartesian domain abstraction*:

$$\wp(\overline{\mathcal{D}}_1 \times \cdots \times \overline{\mathcal{D}}_n) \xleftrightarrow[\alpha^D]{\gamma^D} (\wp(\overline{\mathcal{D}}_1) \times \cdots \times \wp(\overline{\mathcal{D}}_n)) \xleftrightarrow[\alpha_1 \times \cdots \times \alpha_n]{\gamma_1 \times \cdots \times \gamma_n} (\overline{\mathcal{D}}_1^\# \times \cdots \times \overline{\mathcal{D}}_n^\#)$$

and $\wp(\overline{\mathcal{D}}_i) \xleftrightarrow[\alpha_i]{\gamma_i} \overline{\mathcal{D}}_i^\#$ is called the *component abstraction* for domain $\overline{\mathcal{D}}_i$.

4.3.1 Cartesian $(\gamma \circ \alpha)$ -Canonical Propagator Functions

By Lemma 2.17, for a constraint C the smallest sound abstraction of the corresponding exact propagator function C^E on the abstract lattice of a Cartesian domain abstraction is:

$$\widehat{C}^\# = (\alpha_1 \times \cdots \times \alpha_n) \circ \widehat{C}^D \circ (\gamma_1 \times \cdots \times \gamma_n)$$

where $\widehat{C}^D$ is the smallest sound abstraction of C^E on the domain lattice. For a tuple $\langle \mathcal{D}_1^\#, \dots, \mathcal{D}_n^\# \rangle$ where $\mathcal{D}_i^\# \in \overline{\mathcal{D}}_i^\#$ for all $i \in [1, n]$, we have:

$$\begin{aligned} \widehat{C}^\#(\mathcal{D}_1^\#, \dots, \mathcal{D}_n^\#) &= ((\alpha_1 \times \cdots \times \alpha_n) \circ \widehat{C}^D \circ (\gamma_1 \times \cdots \times \gamma_n))(\mathcal{D}_1^\#, \dots, \mathcal{D}_n^\#) \\ &= ((\alpha_1 \times \cdots \times \alpha_n) \circ \widehat{C}^D)(\gamma_1(\mathcal{D}_1^\#), \dots, \gamma_n(\mathcal{D}_n^\#)) \\ &= ((\alpha_1 \times \cdots \times \alpha_n) \circ (\alpha^D \circ C^E \circ \gamma^D)) \left(\gamma_1(\mathcal{D}_1^\#), \dots, \gamma_n(\mathcal{D}_n^\#) \right) \quad [\text{by Lem. 2.16}] \\ &= ((\alpha_1 \times \cdots \times \alpha_n) \circ \alpha^D \circ C^E)(\gamma_1(\mathcal{D}_1^\#) \times \cdots \times \gamma_n(\mathcal{D}_n^\#)) \quad [\text{by (4.3)}] \\ &= (\alpha \circ C^E)(\gamma_1(\mathcal{D}_1^\#) \times \cdots \times \gamma_n(\mathcal{D}_n^\#)) \end{aligned} \tag{4.8}$$

The corresponding $(\gamma \circ \alpha)$ -canonical propagator function is:

$$\begin{aligned}
C^\sharp(\mathcal{D}) &= \mathcal{D} \cap \left(\gamma \circ \widehat{C}^\sharp \circ \alpha \right) (\mathcal{D}) && [\text{by Def. 2.24}] \\
&= \mathcal{D} \cap \left(\gamma \circ \widehat{C}^\sharp \circ (\alpha_1 \times \cdots \times \alpha_n) \circ \alpha^D \right) (\mathcal{D}) && [\text{by Def. 4.3}] \\
&= \mathcal{D} \cap \left(\gamma \circ \widehat{C}^\sharp \circ (\alpha_1 \times \cdots \times \alpha_n) \right) \left(\begin{smallmatrix} \text{dom}(X_1), \dots, \\ \text{dom}(X_n) \end{smallmatrix} \right) && [\text{by (4.2)}] \\
&= \mathcal{D} \cap \left(\gamma \circ \widehat{C}^\sharp \right) (\alpha_1(\text{dom}(X_1)), \dots, \alpha_n(\text{dom}(X_n))) && (4.9)
\end{aligned}$$

A propagator function $C^\sharp$ defined in this way is the weakest propagator function that maintains the consistency induced by the given Cartesian domain abstraction.

4.3.2 Cartesian $\langle \alpha, \gamma \rangle$ -Consistency

The $\langle \alpha, \gamma \rangle$ -consistency induced by a given Cartesian domain abstraction may be expressed in terms of the domains of individual variables, as justified by the following theorem:

Theorem 4.4: Determining Cartesian $\langle \alpha, \gamma \rangle$ -consistency

Consider a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\overline{\mathbf{D}} = \overline{D}_1 \times \cdots \times \overline{D}_n$ of candidate solutions. Let $C \in \mathbf{C}$ be a constraint, and let

$$\wp(\overline{D}_1 \times \cdots \times \overline{D}_n) \xleftrightarrow[\alpha]{\gamma} \left(\overline{D}_1^\sharp \times \cdots \times \overline{D}_n^\sharp \right)$$

be a Cartesian domain abstraction with component abstractions

$$\forall i \in [1, n] : \wp(\overline{D}_i) \xleftrightarrow[\alpha_i]{\gamma_i} \overline{D}_i^\sharp$$

A domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ is $\langle \alpha, \gamma \rangle$ -consistent for C if and only if for each variable X_i such that $i \in [1, n]$, we have $\alpha_i(\text{dom}(X_i)) = \alpha_i(\text{dom}'(X_i))$ where $\text{dom}'(X_i)$ is the set of every value $d \in \text{dom}(X_i)$ such that there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, where an assignment $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if $d_\ell \in (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell))$ for all $\ell \in [1, n]$.

Proof. Let C^E be the exact propagator function implementing the constraint C . By Theorem 3.2, we have

$$\begin{aligned}
\alpha(\mathcal{D}) &= (\alpha \circ C^E \circ \gamma \circ \alpha) (\mathcal{D}) && (4.10) \\
&= (\alpha \circ C^E \circ \gamma \circ (\alpha_1 \times \cdots \times \alpha_n) \circ \alpha^D) (\mathcal{D}) \\
&= (\alpha \circ C^E \circ \gamma \circ (\alpha_1 \times \cdots \times \alpha_n)) (\text{dom}(X_1), \dots, \text{dom}(X_n))
\end{aligned}$$

$$\begin{aligned}
&= (\alpha \circ C^E \circ \gamma) (\alpha_1(\text{dom}(X_1)), \dots, \alpha_n(\text{dom}(X_n))) \\
&= (\alpha \circ C^E \circ \gamma^D \circ (\gamma_1 \times \dots \times \gamma_n)) (\alpha_1(\text{dom}(X_1)), \dots, \alpha_n(\text{dom}(X_n))) \\
&= (\alpha \circ C^E \circ \gamma^D) ((\gamma_1 \circ \alpha_1)(\text{dom}(X_1)), \dots, (\gamma_n \circ \alpha_n)(\text{dom}(X_n))) \\
&= (\alpha \circ C^E) ((\gamma_1 \circ \alpha_1)(\text{dom}(X_1)) \times \dots \times (\gamma_n \circ \alpha_n)(\text{dom}(X_n))) \\
&= \alpha (C \cap ((\gamma_1 \circ \alpha_1)(\text{dom}(X_1)) \times \dots \times (\gamma_n \circ \alpha_n)(\text{dom}(X_n)))) \\
&= ((\alpha_1 \times \dots \times \alpha_n) \circ \alpha^D) \left(C \cap \left((\gamma_1 \circ \alpha_1)(\text{dom}(X_1)) \times \dots \times (\gamma_n \circ \alpha_n)(\text{dom}(X_n)) \right) \right) \\
&= (\alpha_1 \times \dots \times \alpha_n) \left(\left\{ d \in \overline{\mathcal{D}}_1 \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in \\ C \cap \left((\gamma_1 \circ \alpha_1)(\text{dom}(X_1)) \times \dots \right. \\ \left. \times (\gamma_n \circ \alpha_n)(\text{dom}(X_n)) \right) : d_1 = d \end{array} \right\}, \right. \\
&\quad \dots, \\
&\quad \left. \left\{ d \in \overline{\mathcal{D}}_n \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in \\ C \cap \left((\gamma_1 \circ \alpha_1)(\text{dom}(X_1)) \times \dots \right. \\ \left. \times (\gamma_n \circ \alpha_n)(\text{dom}(X_n)) \right) : d_n = d \end{array} \right\} \right) \\
&= (\alpha_1 \times \dots \times \alpha_n) \left(\left\{ d \in \overline{\mathcal{D}}_1 \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in C : \\ d_1 = d \wedge \forall \ell \in [1, n] : \\ d_\ell \in (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell)) \end{array} \right\}, \dots, \right. \\
&\quad \left. \left\{ d \in \overline{\mathcal{D}}_n \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in C : \\ d_n = d \wedge \forall \ell \in [1, n] : \\ d_\ell \in (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell)) \end{array} \right\}, \right) \\
&= \left\langle \alpha_1 \left(\left\{ d \in \overline{\mathcal{D}}_1 \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in C : d_1 = d \wedge \\ \forall \ell \in [1, n] : d_\ell \in (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell)) \end{array} \right\} \right), \dots, \right. \\
&\quad \left. \alpha_n \left(\left\{ d \in \overline{\mathcal{D}}_n \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in C : d_n = d \wedge \\ \forall \ell \in [1, n] : d_\ell \in (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell)) \end{array} \right\} \right) \right\rangle
\end{aligned}$$

Let $\text{dom}'(X_i)$ be the set in the i -th component of this tuple, so that we can write the tuple as $\langle \alpha_1(\text{dom}'(X_1)), \dots, \alpha_n(\text{dom}'(X_n)) \rangle$. Then for all $i \in [1, n]$ we have

$$\begin{aligned}
\text{dom}'(X_i) &\subseteq (\gamma_i \circ \alpha_i)(\text{dom}(X_i)) \\
&\equiv \alpha_i(\text{dom}'(X_i)) \leq^i (\alpha_i \circ \gamma_i \circ \alpha_i)(\text{dom}(X_i)) \\
&\equiv \alpha_i(\text{dom}'(X_i)) \leq^i \alpha_i(\text{dom}(X_i)) \\
&\equiv \text{dom}'(X_i) \subseteq \text{dom}(X_i)
\end{aligned}$$

Therefore, since $\text{dom}(X_i) \subseteq \overline{\mathcal{D}}_i$, we can rewrite $\text{dom}'(X_i)$ as:

$$\text{dom}'(X_i) = \left\{ d \in \text{dom}(X_i) \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in C : d_i = d \wedge \\ \forall \ell \in [1, n] : d_\ell \in (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell)) \end{array} \right\}$$

Thus every $\text{dom}'(X_i)$ meets all the required conditions.

It remains only to show that the abstractions of $\text{dom}(X_i)$ and $\text{dom}'(X_i)$ are equal. Taking the left-hand side of (4.10), we have:

$$\begin{aligned}\alpha(\mathcal{D}) &= ((\alpha_1 \times \cdots \times \alpha_n) \circ \alpha^D)(\mathcal{D}) \\ &= (\alpha_1 \times \cdots \times \alpha_n)(\text{dom}(X_1), \dots, \text{dom}(X_n)) \\ &= \langle \alpha_1(\text{dom}(X_1)), \dots, \alpha_n(\text{dom}(X_n)) \rangle\end{aligned}$$

Hence, we have an equality between n -tuples that we consider component-wise for all $i \in [1, n]$:

$$\alpha_i(\text{dom}(X_i)) = \alpha_i(\text{dom}'(X_i))$$

□

* * *

Theorem 4.4 is relied upon throughout the remainder of the thesis to allow movement back and forth between consistencies defined by conditions on the domains of individual variables and corresponding consistencies defined by Cartesian domain abstractions. For instance, in the next chapter, Theorem 4.4 is used to illustrate the relationship between the *interval approximation* of a variable domain and several versions of a consistency introduced in Chapter 1, namely *bounds consistency*.

5. Bounds Consistencies

As explained in Chapter 1, under *bounds consistency* every current domain *bound* of every variable participates in some solution to the constraint that involves values between the bounds of the domains of the other variables. Propagation to a bounds-consistent domain (Sections 5.2 and 5.3) can be described in abstract interpretation as a calculation on an abstract domain based on interval approximation (Section 5.1).

Section 5.2 presents $\langle \alpha^B, \gamma^B \rangle$ -consistency, a generalized version of bounds consistency that requires only that the domains of the variables it applies to are ordered. For partially-ordered variable domains, Theorem 5.1 shows how to determine when a domain is $\langle \alpha^B, \gamma^B \rangle$ -consistent, a method which is then specialized for totally-ordered variable domains in Theorem 5.4.

Section 5.3 discusses some related consistencies: on the one hand, $\text{bounds}(\mathbb{R})$ consistency, for which there exists a corresponding $\langle \alpha, \gamma \rangle$ -consistency, and on the other hand $\text{bounds}(\mathbb{D})$ consistency and range consistency, for which no corresponding Galois-based consistency exists.

5.1 Interval Approximation

For a lattice $\langle \mathcal{P}, \leq, \bigvee, \bigwedge \rangle$, an *interval* from $\ell \in \mathcal{P}$ to $u \in \mathcal{P}$ is the set of all elements $p \in \mathcal{P}$ satisfying $\ell \leq p \leq u$. The order $\leq$ can be partial for the moment; we specialize to total orders in Section 5.2.2. We distinguish between *concrete intervals*, which are subsets of $\mathcal{P}$ with contiguous elements, and *abstract intervals*, which are pairs of ordered elements from $\mathcal{P}$. For a subset $S \subseteq \mathcal{P}$, we call the concrete interval from the greatest lower bound $\bigwedge S$ to the least upper bound $\bigvee S$ the *interval closure* of S , while we call the abstract interval $[\bigwedge S, \bigvee S]$ the *interval abstraction* of S , enclosing the pair in square brackets instead of the usual angled brackets in order to indicate that it represents an interval in $\mathcal{P}$.

We denote the set of interval abstractions of all $S \subseteq \mathcal{P}$ as $[\mathcal{P}]$, writing $\perp$ to denote the empty abstract interval:

$$[\mathcal{P}] = \{\perp\} \cup \{[\ell, u] \mid \ell, u \in \mathcal{P} \wedge \ell \leq u\}$$

The *interval abstraction function* maps sets to abstract intervals:

$$\alpha^{[\mathcal{P}]}(S) = \begin{cases} \perp & \text{if } S = \emptyset \\ [\bigwedge S, \bigvee S] & \text{otherwise} \end{cases} \quad (5.1)$$

The *interval concretization function* maps abstract intervals to concrete intervals, which are sets:

$$\gamma^{[\mathcal{P}]}(i) = \begin{cases} \emptyset & \text{if } i = \perp \\ \{p \in \mathcal{P} \mid \ell \leq p \leq u\} & \text{if } i = [\ell, u] \end{cases} \quad (5.2)$$

The function $\gamma^{[\mathcal{P}]} \circ \alpha^{[\mathcal{P}]}$ is the *interval closure operation* for $\mathcal{P}$.

From Lemma 2.14 it follows that $\wp(\mathcal{P}) \xleftrightarrow[\alpha^{[\mathcal{P}]}]{\gamma^{[\mathcal{P}]}} [\mathcal{P}]$ is a Galois connection, and hence that $[\mathcal{P}]$ is a complete lattice. The order $\sqsubseteq$, lub $\bigsqcup$, and glb $\bigsqcap$ of this lattice are defined with respect to the power lattice $\wp(\mathcal{P})$:

$$\begin{aligned} \forall i_1, i_2 \in [\mathcal{P}] : \quad & i_1 \sqsubseteq i_2 \text{ if and only if } \gamma^{[\mathcal{P}]}(i_1) \subseteq \gamma^{[\mathcal{P}]}(i_2) \\ \forall \mathcal{I} \subseteq [\mathcal{P}] : \quad & \bigsqcup \mathcal{I} = \alpha^{[\mathcal{P}]} \left(\bigcup_{i \in \mathcal{I}} \gamma^{[\mathcal{P}]}(i) \right) \quad \text{and} \quad \bigsqcap \mathcal{I} = \alpha^{[\mathcal{P}]} \left(\bigcap_{i \in \mathcal{I}} \gamma^{[\mathcal{P}]}(i) \right) \end{aligned}$$

Intuitively, the concrete interval corresponding to $\bigsqcap \mathcal{I}$ is the intersection of the concretization of the abstract intervals in $\mathcal{I}$, and the concrete interval corresponding to $\bigsqcup \mathcal{I}$ is the convex hull of the concretization of the abstract intervals in $\mathcal{I}$.

The interval closure operation is unit approximate, as it maps any singleton set $\{p\} \subseteq \mathcal{P}$ to itself:

$$\begin{aligned} (\gamma^{[\mathcal{P}]} \circ \alpha^{[\mathcal{P}]}) (\{p\}) &= \gamma^{[\mathcal{P}]} \left(\left[\bigwedge \{p\}, \bigvee \{p\} \right] \right) = \gamma^{[\mathcal{P}]}([p, p]) \\ &= \{p' \in \mathcal{P} \mid p \leq p' \leq p\} = \{p\} \end{aligned}$$

The functions $\alpha^{[\mathbb{Z}]}$ and $\gamma^{[\mathbb{Z}]}$, previously encountered in Examples 3.1 to 3.4, are the instances of $\alpha^{[\mathcal{P}]}$ and $\gamma^{[\mathcal{P}]}$ where $\mathcal{P} = \mathbb{Z}$. In that case, the function $\alpha^{[\mathbb{Z}]}$ maps a set of integers to the pair consisting of its minimum and maximum elements, and the function $\gamma^{[\mathbb{Z}]}$ maps such a pair to the set of integers between those values inclusive.

5.2 Bounds Consistency

Bounds consistency, at its most general, corresponds to a Cartesian domain abstraction wherein all component abstractions are Galois connections defining interval approximations:

$$\wp(\overline{\mathcal{D}}_1 \times \cdots \times \overline{\mathcal{D}}_n) \xleftrightarrow[\alpha^{\mathbf{B}}]{\gamma^{\mathbf{B}}} ([\overline{\mathcal{D}}_1] \times \cdots \times [\overline{\mathcal{D}}_n]) \quad (5.3)$$

where the component abstraction for each $i \in [1, n]$ is a Galois connection of the form:

$$\wp(\overline{\mathcal{D}}_i) \xleftrightarrow[\alpha^{[\overline{\mathcal{D}}_i]}]{\gamma^{[\overline{\mathcal{D}}_i]}} [\overline{\mathcal{D}}_i]$$

such that $\alpha^{[\overline{\mathcal{D}}_i]}$ is the interval abstraction function on $\overline{\mathcal{D}}_i$, and $\gamma^{[\overline{\mathcal{D}}_i]}$ is the interval concretization function on $\overline{\mathcal{D}}_i$.

The consistency induced by this Galois connection, $\langle \alpha^B, \gamma^B \rangle$ -consistency, is general enough so as to subsume several common definitions of bounds consistency. We distinguish between two cases: bounds consistency for partially ordered domains is handled in Section 5.2.1, and bounds consistency for totally ordered (e.g., integer) domains is handled in Section 5.2.2.

5.2.1 Bounds Consistency for Partially-Ordered Domains

For domains that are partially ordered the following theorem demonstrates how to determine whether or not a domain is $\langle \alpha^B, \gamma^B \rangle$ -consistent:

Theorem 5.1: Determining $\langle \alpha^B, \gamma^B \rangle$ -consistency (Partial Orders)

Consider a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\overline{\mathbf{D}} = \overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n$ of candidate solutions, where $\langle \overline{\mathcal{D}}_i, \leq^i, \bigwedge^i, \bigvee^i \rangle$ is a lattice with partial order $\leq^i$ for all $i \in [1, n]$. Let $C \in \mathbf{C}$ be a constraint, and let

$$\wp(\overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n) \xrightleftharpoons[\alpha^B]{\gamma^B} ([\overline{\mathcal{D}}_1] \times \dots \times [\overline{\mathcal{D}}_n])$$

be a Cartesian domain abstraction with component abstractions

$$\langle \wp(\overline{\mathcal{D}}_i), \subseteq, \cup, \cap \rangle \xrightleftharpoons[\alpha^{[\overline{\mathcal{D}}_i]}]{\gamma^{[\overline{\mathcal{D}}_i]}} \langle [\overline{\mathcal{D}}_i], \sqsubseteq, \sqcap, \sqcup \rangle$$

for all $i \in [1, n]$.

A domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ is $\langle \alpha^B, \gamma^B \rangle$ -consistent for C if and only if for each variable X_i such that $i \in [1, n]$ we have

$$\bigwedge^i \text{dom}(X_i) = \bigwedge^i \text{dom}'(X_i) \quad \text{and} \quad \bigvee^i \text{dom}(X_i) = \bigvee^i \text{dom}'(X_i)$$

where $\text{dom}'(X_i)$ is the set of all values $d \in \text{dom}(X_i)$ such that there exists a valid tuple $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, where a tuple $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if

$$\bigwedge^\ell \text{dom}(X_\ell) \leq^\ell d_\ell \leq^\ell \bigvee^\ell \text{dom}(X_\ell)$$

for all $\ell \in [1, n]$.

Proof. Let X_i be a variable in $\mathbf{X}$. We will show that the conditions of $\langle \alpha^B, \gamma^B \rangle$ -consistency imposed on $\text{dom}(X_i)$ by Theorem 4.4 are equivalent to those imposed above. In other words, we show that

$$\alpha_i(\text{dom}(X_i)) = \alpha_i(\text{dom}'(X_i)) \quad (5.4a)$$

such that

$$\begin{aligned} & \text{dom}'(X_i) \\ &= \left\{ d \in \text{dom}(X_i) \mid \exists \langle d_1, \dots, d_n \rangle \in C : d_i = d \wedge \forall \ell \in [1, n] : \right. \\ & \quad \left. d_\ell \in (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell)) \right\} \end{aligned} \quad (5.4b)$$

is equivalent to

$$\bigwedge^i \text{dom}(X_i) = \bigwedge^i \text{dom}'(X_i) \text{ and } \bigvee^i \text{dom}(X_i) = \bigvee^i \text{dom}'(X_i) \quad (5.5a)$$

such that

$$\begin{aligned} & \text{dom}'(X_i) \\ &= \left\{ d \in \text{dom}(X_i) \mid \exists \langle d_1, \dots, d_n \rangle \in C : d_i = d \wedge \forall \ell \in [1, n] : \right. \\ & \quad \left. \bigwedge^\ell \text{dom}(X_\ell) \leq^\ell d_\ell \leq^\ell \bigvee^\ell \text{dom}(X_\ell) \right\} \end{aligned} \quad (5.5b)$$

First we show the equivalence of (5.4a) and (5.5a):

$$\begin{aligned} \alpha_i(\text{dom}(X_i)) &= \alpha_i(\text{dom}'(X_i)) \\ &\equiv \langle \bigwedge^i \text{dom}(X_i), \bigvee^i \text{dom}(X_i) \rangle = \langle \bigwedge^i \text{dom}'(X_i), \bigvee^i \text{dom}'(X_i) \rangle \\ &\equiv \bigwedge^i \text{dom}(X_i) = \bigwedge^i \text{dom}'(X_i) \wedge \bigvee^i \text{dom}(X_i) = \bigvee^i \text{dom}'(X_i) \end{aligned}$$

The equivalence of (5.4b) and (5.5b) follows directly from (5.2) and (5.1):

$$\begin{aligned} & (\gamma_\ell \circ \alpha_\ell)(\text{dom}(X_\ell)) \\ &= \gamma_\ell(\bigwedge^\ell \text{dom}(X_\ell), \bigvee^\ell \text{dom}(X_\ell)) \\ &= \left\{ d \in \overline{\mathcal{D}}_i \mid \bigwedge^\ell \text{dom}(X_\ell) \leq^\ell d_\ell \leq^\ell \bigvee^\ell \text{dom}(X_\ell) \right\} \end{aligned}$$

□

Subset Bounds Consistency

In CP, the most widely studied partially (but not totally) ordered domain is that associated with finite set variables [41, 77], that is, variables that take a set, rather than a scalar, as a value. The initial domain of a set variable is a set of sets; we assume, without loss of generality, that the domain is the set of integer sets, $\wp(\mathbb{Z})$. The poset $\langle \wp(\mathbb{Z}), \subseteq \rangle$, ordered by subset inclusion, is a partially ordered set, but not a totally ordered set, as there exist incomparable pairs of elements in $\wp(\mathbb{Z})$ (e.g., neither $\{1, 2, 3\}$ nor $\{2, 3, 4\}$ is a subset of the other). Naturally, $\wp(\mathbb{Z})$ is a complete lattice; in fact, it is the power lattice of $\mathbb{Z}$, or $\langle \wp(\mathbb{Z}), \subseteq, \bigcup, \bigcap \rangle$.

A subset $\mathcal{S}$ of $\wp(\mathbb{Z})$ has an interval abstraction where the lower bound $\mathcal{L}$ is the intersection of the sets of $\mathcal{S}$, and the upper bound $\mathcal{U}$ is the union of the sets of $\mathcal{S}$; furthermore, the interval closure of $\mathcal{S}$ is exactly those sets in $\wp(\wp(\mathbb{Z}))$ that are both supersets of $\mathcal{L}$ and subsets of $\mathcal{U}$. For example, the set $\{\{1\}, \{1, 2\}, \{1, 3\}\}$

has the singleton set $\{1\}$ as its lower bound, the set $\{1, 2, 3\}$ as its upper bound, and the set of sets $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$ as its interval closure.

There is a clear similarity between the interval closure $[\wp(\mathbb{Z})]$ and the well-known bounds consistency for set variables known as subset bounds consistency:

Definition 5.2: Subset bounds consistency [14, 41]

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\bar{\mathbf{D}} = \bar{\mathcal{D}}_1 \times \dots \times \bar{\mathcal{D}}_n$ of candidate solutions, where $\bar{\mathcal{D}}_i$ is a finite subset of $\wp(\mathbb{Z})$ for all $i \in [1, n]$. Let $C \in \mathbf{C}$ be a constraint.

A domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ is *subset bounds consistent* for C if and only if for each set variable X_i such that $i \in [1, n]$ we have

$$\bigcap \text{dom}(X_i) = \bigcap \text{dom}'(X_i) \quad \text{and} \quad \bigcup \text{dom}(X_i) = \bigcup \text{dom}'(X_i)$$

where $\text{dom}'(X_i)$ is the set of all values in $d \in \text{dom}(X_i)$ such that there exists a valid tuple $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, where a tuple $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if

$$\bigcap \text{dom}(X_\ell) \subseteq d_\ell \subseteq \bigcup \text{dom}(X_\ell)$$

for all $\ell \in [1, n]$.

Subset bounds consistency is induced by a Galois connection between the powerset lattice of $\wp(\mathbb{Z})$ and the interval lattice of $\wp(\mathbb{Z})$, namely the Cartesian domain abstraction

$$\wp(\bar{\mathcal{D}}_1 \times \dots \times \bar{\mathcal{D}}_n) \xleftrightarrow[\alpha^{\text{B}(\subseteq)}]{\gamma^{\text{B}(\subseteq)}} ([\bar{\mathcal{D}}_1] \times \dots \times [\bar{\mathcal{D}}_n]) \quad (5.6)$$

where for all $i \in [1, n]$ the set $\bar{\mathcal{D}}_i$ is a finite subset of $\wp(\mathbb{Z})$ and the component abstraction for each i is a Galois connection of the form

$$\begin{aligned} \wp(\wp(\mathbb{Z})) &\xleftrightarrow[\alpha^{[\wp(\mathbb{Z})]}]{\gamma^{[\wp(\mathbb{Z})]}} [\wp(\mathbb{Z})] \\ \alpha^{[\wp(\mathbb{Z})]}(\mathcal{S}) &= \begin{cases} \perp & \text{if } \mathcal{S} = \emptyset \\ [\bigcap \mathcal{S}, \bigcup \mathcal{S}] & \text{otherwise} \end{cases} \\ \text{where} \quad \gamma^{[\wp(\mathbb{Z})]}(I) &= \begin{cases} \emptyset & \text{if } I = \perp \\ \{\mathcal{S} \in \wp(\mathbb{Z}) \mid \mathcal{L} \subseteq \mathcal{S} \subseteq \mathcal{U}\} & \text{if } I = [\mathcal{L}, \mathcal{U}] \end{cases} \end{aligned} \quad (5.7)$$

That the consistency induced by this Galois connection is the same as subset bounds consistency is a direct consequence of Theorem 5.1:

Proposition 5.3

$\langle \alpha^{\text{B}(\subseteq)}, \gamma^{\text{B}(\subseteq)} \rangle$ -consistency is equivalent to subset bounds consistency.

Proof. From Theorem 5.1 and (5.7), a domain $\mathcal{D} \in \overline{\mathbf{D}}$ is $\langle \alpha^{\mathbf{B}(\subseteq)}, \gamma^{\mathbf{B}(\subseteq)} \rangle$ -consistent if and only if for each variable X_i such that $i \in [1, n]$ we have:

$$\bigcap \text{dom}(X_i) = \bigcap \text{dom}'(X_i) \text{ and } \bigcup \text{dom}(X_i) = \bigcup \text{dom}'(X_i)$$

such that

$$\text{dom}'(X_i) = \left\{ d \in \text{dom}(X_i) \mid \exists \langle d_1, \dots, d_n \rangle \in C : d_i = d \wedge \forall \ell \in [1, n] : \bigcap \text{dom}(X_\ell) \subseteq d_\ell \subseteq \bigcup \text{dom}(X_\ell) \right\}$$

Hence, the conditions on domains that are $\langle \alpha^{\mathbf{B}(\subseteq)}, \gamma^{\mathbf{B}(\subseteq)} \rangle$ -consistent are the same on those that are subset bounds consistent. $\square$

Definition 5.2 applies to problems where all the variables are set variables, but bounds consistency for set variables is often defined as a consistency applying to a mixture of set and integer variables. We call this latter consistency, which we address in Section 6.1, *set-integer bounds consistency*.

5.2.2 Bounds Consistency for Totally-Ordered Domains

As the conditions supplied by Theorem 5.1 for determining $\langle \alpha^{\mathbf{B}}, \gamma^{\mathbf{B}} \rangle$ -consistency apply to all partially ordered variable domains, those conditions also hold for totally ordered domains, e.g., when $\overline{\mathcal{D}} \subset \mathbb{Z}$. However, unlike for a partially-ordered domain, for a totally-ordered variable domain $\text{dom}(X_i)$ it is not necessary to construct the whole set of elements in $\text{dom}(X_i)$ that participate in a valid solution as defined in Theorem 5.1. Instead, it is sufficient to determine whether the maximum and minimum elements of $\text{dom}(X_i)$ participate in a valid solution.

If a poset $\mathcal{P}$ is a total order, then for any subset $\mathcal{S}$ of $\mathcal{P}$ the lub (respectively, glb) of $\mathcal{S}$ is the maximum (respectively, minimum) element of $\mathcal{S}$. This property does not hold for partial orders in general: for a subset $\mathcal{S}$ of a partial order $\mathcal{P}$, the lub and glb of $\mathcal{S}$ might not be members of $\mathcal{S}$. From this property it follows that the domain $\text{dom}(X) \subseteq \overline{\mathcal{D}}$ of the variable X , where $\overline{\mathcal{D}}$ is totally ordered, is bounds consistent for a constraint C if and only if both the maximum and minimum value of $\text{dom}(X)$ participate in an assignment satisfying C composed of values from the interval approximations of the domains of the remaining variables. In other words, Theorem 5.1 may be rewritten for totally-ordered domains as follows:

Theorem 5.4: Determining $\langle \alpha^B, \gamma^B \rangle$ -consistency (Total Orders)

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\bar{\mathbf{D}} = \bar{D}_1 \times \dots \times \bar{D}_n$ of candidate solutions, where $\langle \bar{D}_i, \leq^i, \bigwedge^i, \bigvee^i \rangle$ is a lattice with total order $\leq^i$ for all $i \in [1, n]$. Let $C \in \mathbf{C}$ be a constraint, and let

$$\wp(\bar{D}_1 \times \dots \times \bar{D}_n) \xleftrightarrow[\alpha^B]{\gamma^B} ([\bar{D}_1] \times \dots \times [\bar{D}_n])$$

be a Cartesian domain abstraction with component abstractions

$$\langle \wp(\bar{D}_i), \subseteq, \bigcup, \bigcap \rangle \xleftrightarrow[\alpha^{[\bar{D}_i]}]{\gamma^{[\bar{D}_i]}} \langle [\bar{D}_i], \sqsubseteq, \sqcap, \sqcup \rangle$$

for all $i \in [1, n]$.

A domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ is $\langle \alpha^B, \gamma^B \rangle$ -consistent for C if and only if for each variable X_i such that $i \in [1, n]$, and for each value d such that

$$d \in \{\min \text{dom}(X_i), \max \text{dom}(X_i)\}$$

there exists a valid tuple $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, where a tuple $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if

$$\min(X_\ell) \leq^\ell d_\ell \leq^\ell \max(X_\ell)$$

for all $\ell \in [1, n]$.

Proof. Let X_i be a variable in $\mathbf{X}$, and let $\text{dom}'(X_i)$ be the set:

$$\begin{aligned} & \text{dom}'(X_i) \\ &= \left\{ d \in \text{dom}(X_i) \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in C : \\ d_i = d \wedge \forall \ell \in [1, n] : \\ \bigwedge^\ell \text{dom}(X_\ell) \leq^\ell d_\ell \leq^\ell \bigvee^\ell \text{dom}(X_\ell) \end{array} \right\} \end{aligned} \quad (5.8)$$

Since $\leq^i$ is a total order, we also have:

$$\begin{aligned} & \text{dom}'(X_i) \\ &= \left\{ d \in \text{dom}(X_i) \mid \begin{array}{l} \exists \langle d_1, \dots, d_n \rangle \in C : \\ d_i = d \wedge \forall \ell \in [1, n] : \\ \min \text{dom}(X_\ell) \leq^\ell d_\ell \leq^\ell \max \text{dom}(X_\ell) \end{array} \right\} \end{aligned} \quad (5.9)$$

We will show that the conditions of $\langle \alpha^B, \gamma^B \rangle$ -consistency imposed on $\text{dom}(X_i)$ by Theorem 5.1 are equivalent to

$$\begin{aligned} & \exists \langle d_1, \dots, d_n \rangle \in C : \\ & d_i = d \wedge \forall \ell \in [1, n] : \min \text{dom}(X_\ell) \leq^\ell d_\ell \leq^\ell \max \text{dom}(X_\ell) \end{aligned} \quad (5.10)$$

for all $d \in \{\min \text{dom}(X_i), \max \text{dom}(X_i)\}$, i.e., the conditions imposed above.

$\Rightarrow$: Assume that $\mathcal{D}$ is $\langle \alpha^B, \gamma^B \rangle$ -consistent. From Theorem 5.1 it follows that $\bigwedge_i \text{dom}(X_i)$ is equal to $\bigwedge_i \text{dom}'(X_i)$. Further, since $\leq^i$ is a total order, $\max \text{dom}(X_i)$ is equal to $\max \text{dom}'(X_i)$. Hence, $\max \text{dom}(X_i)$ is an element of $\text{dom}'(X_i)$, so by (5.9) there exists a tuple $\langle d_1, \dots, d_n \rangle \in C$ where $\min(X_\ell) \leq^\ell d_\ell \leq^\ell \max(X_\ell)$ for all $\ell \in [1, n]$, with $d_i = \max \text{dom}(X_i)$. By a similar argument, there exists an assignment in $\text{dom}'(X_i)$ with $d_i = \min \text{dom}(X_i)$. Hence, (5.10) is satisfied.

$\Leftarrow$: Assume (5.10) holds, so there exists an assignment $a = \langle d_1, \dots, d_n \rangle$ in C with $d_i = \min \text{dom}(X_i)$ such that $\min(X_\ell) \leq^\ell d_\ell \leq^\ell \max(X_\ell)$ for all $\ell \in [1, n]$. By (5.9), $\min \text{dom}(X_i) \in \text{dom}'(X_i)$, and since $\text{dom}'(X_i) \subseteq \text{dom}(X_i)$ we have that $\min \text{dom}(X_i)$ must be the minimum element of $\text{dom}'(X_i)$. Therefore, $\bigvee_i \text{dom}(X_i) = \bigvee_i \text{dom}'(X_i)$; and by a similar argument $\bigwedge_i \text{dom}(X_i) = \bigwedge_i \text{dom}'(X_i)$. Hence, by Theorem 5.1, $\mathcal{D}$ is $\langle \alpha^B, \gamma^B \rangle$ -consistent. $\square$

Bounds($\mathbb{Z}$) Consistency

A common form of bounds consistency for integer variables, in which every current domain bound of every variable participates in some solution to the constraint that involves integer values between the current domain bounds of the other variables, was previously introduced in Chapter 1. This consistency, in order to distinguish it from other closely related definitions of bounds consistency (discussed below), is sometimes called bounds($\mathbb{Z}$) consistency:

Definition 5.5: Bounds($\mathbb{Z}$) Consistency [21, 101]

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\bar{\mathbf{D}} = \bar{D}_1 \times \dots \times \bar{D}_n$ of candidate solutions, where $\bar{D}_i$ is a finite subset of $\mathbb{Z}$ for all $i \in [1, n]$. Let $C \in \mathbf{C}$ be a constraint.

A domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ is *bounds($\mathbb{Z}$) consistent* for C if and only if for each variable X_i such that $i \in [1, n]$, and for each value d such that

$$d \in \{\min \text{dom}(X_i), \max \text{dom}(X_i)\}$$

there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$ where an assignment $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if d_ℓ is an integer value such that

$$\min \text{dom}(X_\ell) \leq d_\ell \leq \max \text{dom}(X_\ell)$$

for all $\ell \in [1, n]$.

Bounds($\mathbb{Z}$) consistency is induced by a Galois connection between the power-set lattice of $\mathbb{Z}$ and the interval lattice of $\mathbb{Z}$, namely the Cartesian domain abstraction

$$\wp(\bar{D}_1 \times \dots \times \bar{D}_n) \xleftrightarrow[\alpha^{B(\mathbb{Z})}]{\gamma^{B(\mathbb{Z})}} ([\bar{D}_1] \times \dots \times [\bar{D}_n]) \quad (5.11)$$

where for all $i \in [1, n]$ the set $\overline{\mathcal{D}}_i$ is a finite subset of $\mathbb{Z}$ and the component abstraction for i is a Galois connection of the following form:

$$\wp(\mathbb{Z}) \xleftrightarrow[\alpha_i^{[\mathbb{Z}]}]{\gamma_i^{[\mathbb{Z}]}} [\mathbb{Z}] \text{ where } \alpha_i^{[\mathbb{Z}]}(\mathcal{S}) = [\min \mathcal{S}, \max \mathcal{S}] \quad (5.12)$$

$$\gamma_i^{[\mathbb{Z}]}([\ell, u]) = \{d \in \mathbb{Z} \mid d \in [\ell, u]\}$$

That the consistency induced by this Galois connection is $\text{bounds}(\mathbb{Z})$ consistency is a direct consequence of Theorem 5.4:

Proposition 5.6

$\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistency is equivalent to $\text{bounds}(\mathbb{Z})$ consistency.

Proof. From Theorem 5.4 and (5.11), a domain $\mathcal{D} \in \overline{\mathbf{D}}$ is $\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistent if and only if for each variable X_i such that $i \in [1, n]$ we have

$$\exists \langle d_1, \dots, d_n \rangle \in C : d_i = d \wedge \forall \ell \in [1, n] : \min \text{dom}(X_\ell) \leq d_\ell \leq \max \text{dom}(X_\ell) \quad (5.13)$$

for all $d \in \{\min \text{dom}(X_i), \max \text{dom}(X_i)\}$. Hence, the conditions on domains that are $\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistent are identical to those on domains that are $\text{bounds}(\mathbb{Z})$ consistent. $\square$

Example 5.1

The smallest sound abstraction $\widehat{\text{ALLDIFFERENT}}^D$ on the domain lattice of the exact propagator function ALLDIFFERENT^E for the ALLDIFFERENT constraint was defined in Example 4.1. By (4.8), the smallest sound abstraction of ALLDIFFERENT^E on the lattice of integer interval approximations is:

$$\begin{aligned} & \widehat{\text{ALLDIFFERENT}}^{B(\mathbb{Z})}(\mathcal{I}_1, \dots, \mathcal{I}_n) \\ &= \left(\left(\begin{pmatrix} \alpha_1^{[\mathbb{Z}]} \times \\ \dots \\ \times \alpha_n^{[\mathbb{Z}]} \end{pmatrix} \circ \widehat{\text{ALLDIFFERENT}}^D \circ \begin{pmatrix} \gamma_1^{[\mathbb{Z}]} \times \\ \dots \\ \times \gamma_n^{[\mathbb{Z}]} \end{pmatrix} \right) (\mathcal{I}_1, \dots, \mathcal{I}_n) \right) \\ &= \left(\left(\begin{pmatrix} \alpha_1^{[\mathbb{Z}]} \times \\ \dots \\ \times \alpha_n^{[\mathbb{Z}]} \end{pmatrix} \circ \widehat{\text{ALLDIFFERENT}}^D \right) \left(\begin{pmatrix} \gamma_1^{[\mathbb{Z}]}(\mathcal{I}_1), \\ \dots, \\ \gamma_n^{[\mathbb{Z}]}(\mathcal{I}_n) \end{pmatrix} \right) \right) \\ &= \left(\begin{pmatrix} \alpha_1^{[\mathbb{Z}]} \times \\ \dots \\ \times \alpha_n^{[\mathbb{Z}]} \end{pmatrix} \left(\left\{ \begin{array}{l} v_1 \in \gamma_1^{[\mathbb{Z}]}(\mathcal{I}_1) \mid \exists \langle v_1, \dots, v_n \rangle \in \{v_1\} \times \\ \gamma_2^{[\mathbb{Z}]}(\mathcal{I}_2) \times \dots \times \gamma_n^{[\mathbb{Z}]}(\mathcal{I}_n) : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\}, \dots, \right. \right. \\ & \quad \left. \left. \left\{ \begin{array}{l} v_n \in \gamma_n^{[\mathbb{Z}]}(\mathcal{I}_n) \mid \exists \langle v_1, \dots, v_n \rangle \in \gamma_1^{[\mathbb{Z}]}(\mathcal{I}_1) \times \\ \dots \times \gamma_{n-1}^{[\mathbb{Z}]}(\mathcal{I}_{n-1}) \times \{v_n\} : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right) \right) \end{aligned}$$

$$= \left\langle \alpha_1^{[\mathbb{Z}]} \left(\left\{ v_1 \in \gamma_1^{[\mathbb{Z}]}(\mathcal{I}_1) \mid \begin{array}{l} \exists \langle v_1, \dots, v_n \rangle \in \{v_1\} \times \\ \gamma_2^{[\mathbb{Z}]}(\mathcal{I}_2) \times \dots \times \gamma_n^{[\mathbb{Z}]}(\mathcal{I}_n) : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right), \dots, \right. \\ \left. \alpha_n^{[\mathbb{Z}]} \left(\left\{ v_n \in \gamma_n^{[\mathbb{Z}]}(\mathcal{I}_n) \mid \begin{array}{l} \exists \langle v_1, \dots, v_n \rangle \in \gamma_1^{[\mathbb{Z}]}(\mathcal{I}_1) \times \\ \dots \times \gamma_{n-1}^{[\mathbb{Z}]}(\mathcal{I}_{n-1}) \times \{v_n\} : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right) \right\rangle$$

We now derive the n -ary $(\gamma^{B(\mathbb{Z})} \circ \alpha^{B(\mathbb{Z})})$ -canonical propagator function for ALLDIFFERENT. By (4.9), we have:

$$\text{ALLDIFFERENT}^{B(\mathbb{Z})}(\mathcal{D}) \\ = \mathcal{D} \cap \left(\gamma^{B(\mathbb{Z})} \circ \widehat{\text{ALLDIFFERENT}}^{B(\mathbb{Z})} \right) \begin{pmatrix} \alpha_1^{[\mathbb{Z}]}(\text{dom}(X_1)), \\ \dots, \\ \alpha_n^{[\mathbb{Z}]} \text{dom}(X_n) \end{pmatrix}$$

Writing ϕ for the integer interval closure operation $\gamma^{[\mathbb{Z}]} \circ \alpha^{[\mathbb{Z}]}$ gives:

$$\text{ALLDIFFERENT}^{B(\mathbb{Z})}(\mathcal{D}) \\ = \mathcal{D} \cap \gamma^{B(\mathbb{Z})} \left(\alpha_1^{[\mathbb{Z}]} \left(\left\{ v_1 \in \phi(\text{dom}(X_1)) \mid \begin{array}{l} \exists \langle v_1, \dots, v_n \rangle \in \{v_1\} \times \\ \phi(\text{dom}(X_2)) \times \dots \times \\ \phi(\text{dom}(X_n)) : \\ \forall i, j \in [1, n], i \neq j : \\ v_i \neq v_j \end{array} \right\} \right), \dots, \right. \\ \left. \alpha_n^{[\mathbb{Z}]} \left(\left\{ v_n \in \phi(\text{dom}(X_n)) \mid \begin{array}{l} \exists \langle v_1, \dots, v_n \rangle \in \\ \phi(\text{dom}(X_1)) \times \dots \times \\ \phi(\text{dom}(X_{n-1})) \times \{v_n\} : \\ \forall i, j \in [1, n], i \neq j : \\ v_i \neq v_j \end{array} \right\} \right) \right) \\ = \mathcal{D} \cap \gamma^D \left(\phi \left(\left\{ v_1 \in \phi(\text{dom}(X_1)) \mid \begin{array}{l} \exists \langle v_1, \dots, v_n \rangle \in \{v_1\} \times \\ \phi(\text{dom}(X_2)) \times \dots \times \\ \phi(\text{dom}(X_n)) : \\ \forall i, j \in [1, n], i \neq j : \\ v_i \neq v_j \end{array} \right\} \right), \right. \\ \dots, \\ \left. \phi \left(\left\{ v_n \in \phi(\text{dom}(X_n)) \mid \begin{array}{l} \exists \langle v_1, \dots, v_n \rangle \in \\ \phi(\text{dom}(X_1)) \times \dots \times \\ \phi(\text{dom}(X_{n-1})) \times \{v_n\} : \\ \forall i, j \in [1, n], i \neq j : \\ v_i \neq v_j \end{array} \right\} \right) \right)$$

$$\begin{aligned}
&= \mathcal{D} \cap \phi \left(\left\{ \begin{array}{l} v_1 \in \phi(\text{dom}(X_1)) \\ \exists \langle v_1, \dots, v_n \rangle \in \{v_1\} \times \phi(\text{dom}(X_2)) \times \dots \times \phi(\text{dom}(X_n)) : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \times \dots \right. \\
&\quad \times \phi \left(\left\{ \begin{array}{l} v_n \in \phi(\text{dom}(X_n)) \\ \exists \langle v_1, \dots, v_n \rangle \in \phi(\text{dom}(X_1)) \times \dots \times \phi(\text{dom}(X_{n-1})) \times \{v_n\} : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right)
\end{aligned}$$

Consider once more the starting domain from Example 2.1, namely $\{1, 2\} \times \{1, 2\} \times \{2, 3, 4, 5\} \times \{4\}$. The interval closures of the sets making up this Cartesian product are $\phi(\{1, 2\}) = [1, 2]$, $\phi(\{2, 3, 4, 5\}) = [2, 5]$, and $\phi(\{4\}) = [4, 4]$. Applying $\text{ALLDIFFERENT}^{\text{B}(\mathbb{Z})}$ gives:

$$\begin{aligned}
&\text{ALLDIFFERENT}^{\text{B}(\mathbb{Z})}(\{1, 2\} \times \{1, 2\} \times \{2, 3, 4, 5\} \times \{4\}) \\
&= \{1, 2\} \times \{1, 2\} \times \{2, 3, 4, 5\} \times \{4\} \cap \\
&\quad \phi \left(\left\{ \begin{array}{l} v_1 \in [1, 2] \\ \exists \langle v_1, \dots, v_4 \rangle \in \{v_1\} \times [1, 2] \times [2, 5] \times [4, 4] : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right) \times \\
&\quad \phi \left(\left\{ \begin{array}{l} v_2 \in [1, 2] \\ \exists \langle v_1, \dots, v_4 \rangle \in [1, 2] \times \{v_2\} \times [2, 5] \times [4, 4] : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right) \times \\
&\quad \phi \left(\left\{ \begin{array}{l} v_3 \in [2, 5] \\ \exists \langle v_1, \dots, v_4 \rangle \in [1, 2] \times [1, 2] \times \{v_3\} \times [4, 4] : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right) \times \\
&\quad \phi \left(\left\{ \begin{array}{l} v_4 \in [4, 4] \\ \exists \langle v_1, \dots, v_4 \rangle \in [1, 2] \times [1, 2] \times [2, 5] \times \{v_4\} : \\ \forall i, j \in [1, n], i \neq j : v_i \neq v_j \end{array} \right\} \right) \\
&= \{1, 2\} \times \{1, 2\} \times \{2, 3, 4, 5\} \times \{4\} \cap \\
&\quad \phi(\{1, 2\}) \times \phi(\{1, 2\}) \times \phi(\{2, 3, 4, 5\}) \times \phi(\{4\}) \\
&= \{1, 2\} \times \{1, 2\} \times \{2, 3, 4, 5\} \times \{4\} \cap [1, 2] \times [1, 2] \times [2, 5] \times [4, 4] \\
&= \{1, 2\} \times \{1, 2\} \times \{3, 4, 5\} \times \{4\}
\end{aligned}$$

So from to this starting domain the propagator function $\widehat{\text{ALLDIFFERENT}}^{\text{B}(\mathbb{Z})}$ yields exactly the result that would be expected from Example 2.1.

5.3 Other Forms of Bounds Consistency

$\text{Bounds}(\mathbb{Z})$ consistency is not the only bounds consistency for integer variables. Other variants include $\text{bounds}(\mathbb{R})$ consistency [21, 87], $\text{bounds}(\text{D})$ consistency [21, 31], and range consistency [59]. These three consistencies can-

Figure 5.1. Implication graph for consistencies covered in Sections 4.2, 5.2.2, and 5.3.

not be expressed as $\langle \alpha^B, \gamma^B \rangle$ -consistency, but each is related to $\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistency: $\text{bounds}(\mathbb{R})$ consistency may be defined as a further abstraction of $\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistency, while $\text{bounds}(\mathbb{D})$ and range consistency are domain approximations similar to $\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistency. The relationship of all of these consistencies is illustrated in Figure 5.1.

In Section 5.3.1 we define a Cartesian domain abstraction that is equivalent to $\text{bounds}(\mathbb{R})$ consistency, and in Section 5.3.2 we demonstrate why $\text{bounds}(\mathbb{D})$ consistency and range consistency are not expressible within our Galois connection based framework.

5.3.1 $\text{Bounds}(\mathbb{R})$ Consistency

For some constraints, it is computationally less expensive to prune a current domain by considering the *real-number* intervals corresponding to the integer interval approximation of the current domain of each variable, rather than the integer intervals themselves [87]. The following consistency, which applies to propagator functions that reason on real-number intervals, is implied by $\text{bounds}(\mathbb{Z})$ consistency, as shown in Figure 5.1. Note that we write $[\ell, u]^{\mathbb{R}}$ to mean the set of all *real-number* values d such that $\ell \leq d \leq u$.

Definition 5.7: $\text{Bounds}(\mathbb{R})$ Consistency [21, 87]

Consider a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\overline{\mathbf{D}} = \overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n$ of candidate solutions, where $\overline{\mathcal{D}}_i$ is a finite subset of $\mathbb{Z}$ for all $i \in [1, n]$. Let $C \in \mathbf{C}$ be a constraint.

A domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ is *$\text{bounds}(\mathbb{R})$ consistent* for C if and only if for each variable X_i such that $i \in [1, n]$, and for each value d such that

$$d \in \{\min \text{dom}(X_i), \max \text{dom}(X_i)\}$$

there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, where an assignment $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if d_ℓ is a *real-number* value such that

$$d_\ell \in [\min \text{dom}(X_\ell), \max \text{dom}(X_\ell)]^{\mathbb{R}}$$

for all $\ell \in [1, n]$.

$\text{Bounds}(\mathbb{R})$ consistency is induced by the Cartesian domain abstraction

$$\wp(\overline{\mathcal{D}}_1 \times \cdots \times \overline{\mathcal{D}}_n) \xleftrightarrow[\alpha^{\text{B}(\mathbb{R})}]{\gamma^{\text{B}(\mathbb{R})}} [\overline{\mathcal{D}}_1]^\mathbb{R} \times \cdots \times [\overline{\mathcal{D}}_n]^\mathbb{R} \quad (5.14)$$

where for all $i \in [1, n]$ the set $\overline{\mathcal{D}}_i$ is a finite subset of $\mathbb{Z}$, and the component abstraction for i is a Galois connection constructed via composition with the functions $\alpha_i^{[\mathbb{Z}]}$ and $\gamma_i^{[\mathbb{Z}]}$ from (5.12), as follows:

$$\wp(\mathbb{Z}) \xleftrightarrow[\alpha_i^{[\mathbb{Z}]}]{\gamma_i^{[\mathbb{Z}]}} [\mathbb{Z}] \xleftrightarrow[\alpha_i^{[\mathbb{Z}]^\mathbb{R}}]{\gamma_i^{[\mathbb{Z}]^\mathbb{R}}} [\mathbb{Z}]^\mathbb{R} \text{ where } \begin{aligned} \alpha_i^{[\mathbb{Z}]^\mathbb{R}}([\ell, u]) &= [\ell, u]^\mathbb{R} \\ \gamma_i^{[\mathbb{Z}]^\mathbb{R}}([\ell, u]^\mathbb{R}) &= [[\ell], [u]] \end{aligned} \quad (5.15)$$

The equivalence of $\langle \alpha^{\text{B}(\mathbb{R})}, \gamma^{\text{B}(\mathbb{R})} \rangle$ -consistency and $\text{bounds}(\mathbb{R})$ consistency follows directly:

Proposition 5.8

$\langle \alpha^{\text{B}(\mathbb{R})}, \gamma^{\text{B}(\mathbb{R})} \rangle$ -consistency is equivalent to $\text{bounds}(\mathbb{R})$ consistency.

Proof. Follows directly from Theorems 4.4 and 5.4. □

5.3.2 $\text{Bounds}(\text{D})$ Consistency and Range Consistency

The two other known integer bounds consistencies, $\text{bounds}(\text{D})$ and range consistency, have definitions similar to that of $\text{bounds}(\mathbb{Z})$ consistency:

Definition 5.9: $\text{Bounds}(\text{D})$ Consistency [21, 31]

Consider a CSP $\langle \mathbf{X}, \overline{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\overline{\mathbf{D}} = \overline{\mathcal{D}}_1 \times \cdots \times \overline{\mathcal{D}}_n$ of candidate solutions, where $\overline{\mathcal{D}}_i$ is a finite subset of $\mathbb{Z}$ for all $i \in [1, n]$. Let $C \in \mathbf{C}$ be a constraint.

A domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ is *bounds(D) consistent* for C if and only if for each variable X_i such that $i \in [1, n]$, and each value d such that

$$d \in \{\min \text{dom}(X_i), \max \text{dom}(X_i)\}$$

there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, where an assignment $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if

$$d_\ell \in \text{dom}(X_\ell)$$

for all $\ell \in [1, n]$.

Definition 5.10: Range Consistency [59]

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\bar{\mathbf{D}} = \bar{D}_1 \times \dots \times \bar{D}_n$ of candidate solutions, where $\bar{D}_i$ is a finite subset of $\mathbb{Z}$ for all $i \in [1, n]$. Let $C \in \mathbf{C}$ be a constraint.

A domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ is *range consistent* for C if and only if for each variable X_i such that $i \in [1, n]$, and each value d such that

$$d \in \text{dom}(X_i)$$

there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, where an assignment $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if d_ℓ is an integer value such that

$$d_\ell \in [\min \text{dom}(X_\ell), \max \text{dom}(X_\ell)]$$

for all $\ell \in [1, n]$.

While Definition 5.5 appears quite similar to Definitions 5.9 and 5.10, the latter two consistencies are significantly different from the rest of the consistencies addressed in this thesis. The difference may be understood by considering there is no ϕ -completeness property for any closure operation ϕ that corresponds to either of these consistencies. Instead, each has a corresponding completeness property of a similar type, called $\phi\psi$ -completeness, whose specialization $\phi\phi$ -completeness is ϕ -completeness:

Definition 5.11: $\phi\psi$ -completeness [26, 94]

Given variables $\mathbf{X}$ with initial domain $\bar{\mathbf{D}} = \bar{D}_1 \times \dots \times \bar{D}_n$, a constraint C , and closure operations $\phi: \bar{\mathbf{D}} \rightarrow \bar{\mathbf{D}}$ and $\psi: \bar{\mathbf{D}} \rightarrow \bar{\mathbf{D}}$, a propagator function p is $\phi\psi$ -complete if and only if for any current domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ we have $p(\mathcal{D}) \subseteq \psi(C \cap \phi(\mathcal{D}))$.

For constraint C with an exact propagator function C^E , a propagator function $C^\#$ implementing C is *bounds(D)-complete* [94] if and only if:

$$C^D(\mathcal{D}) \subseteq ((\gamma^D \circ \alpha^D) \circ C^E \circ (\gamma^B \circ \alpha^B))(\mathcal{D}) \quad (5.16)$$

Similarly, $C^\#$ is *range-complete* [94] if and only if:

$$C^R(\mathcal{D}) \subseteq ((\gamma^B \circ \alpha^B) \circ C^E \circ (\gamma^D \circ \alpha^D))(\mathcal{D}) \quad (5.17)$$

Figure 5.2 illustrates the relationship of bounds(D)-completeness and range-completeness to both bounds($\mathbb{Z}$)-completeness and domain-completeness. A domain-complete propagator function C^D implementing a constraint C computes the smallest sound approximation $\hat{C}^D$ on the domain lattice $\wp(\bar{D}_1) \times \dots \times \wp(\bar{D}_n)$ of the exact propagator function C^E . A bounds($\mathbb{Z}$)-complete

Figure 5.2. Comparison of domain-complete, bounds(D)-complete, range-complete, and bounds($\mathbb{Z}$)-complete propagation, where the function C^D computes the upper bound $\gamma^D \circ \alpha^D \circ C^E \circ \gamma^D \circ \alpha^D$ of those domains returned by a domain-complete propagator function for the constraint C , and the functions $\alpha = \alpha_1^{[\mathbb{Z}]} \times \dots \times \alpha_n^{[\mathbb{Z}]}$ and $\gamma = \gamma_1^{[\mathbb{Z}]} \times \dots \times \gamma_n^{[\mathbb{Z}]}$ are the Cartesian products of the component abstractions of the Cartesian domain abstraction defining $\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistency (5.11).

propagator function $C^{B(\mathbb{Z})}$ over-approximates the computation of $\widehat{C}^D$ on the domain lattice by instead computing the smallest sound approximation of $\widehat{C}^D$ on the bounds($\mathbb{Z}$) lattice $[\mathcal{D}_1] \times \cdots \times [\mathcal{D}_n]$. In contrast, neither a bounds(D)-complete nor a range-complete propagator function over-approximates $\widehat{C}^D$: both prune the domain by computing $\widehat{C}^D$ on the domain lattice, but a range-complete propagator function over-approximates the initial domain, while a bounds(D)-complete propagator function over-approximates the resulting domain. The over-approximation in both cases is the closure operator induced by the Cartesian product of the component abstractions $\alpha_1^{[\mathbb{Z}]} \times \cdots \times \alpha_n^{[\mathbb{Z}]}$ and $\gamma_1^{[\mathbb{Z}]} \times \cdots \times \gamma_n^{[\mathbb{Z}]}$ of the Cartesian domain abstraction defining $\langle \alpha^{B(\mathbb{Z})}, \gamma^{B(\mathbb{Z})} \rangle$ -consistency (5.11). The outcome in both cases is a pruning strength somewhere between domain consistency and bounds($\mathbb{Z}$) consistency: domain consistency implies both bounds(D) consistency and range consistency, both of which in turn imply bounds($\mathbb{Z}$) consistency. Bounds(D) consistency and range consistency are themselves incomparable, however (see, e.g., [26] for an example).

Theorem 3.5, which has to this point supplied the connection between completeness and consistency, does not apply here: neither bounds(D) -completeness nor range completeness can be defined in terms of the smallest sound abstraction of an exact propagator function. In fact, a bounds(D)-complete propagator function does maintain bounds(D) consistency, and similarly a range-complete propagator does maintain range consistency [94], but it is currently unclear under what circumstances these results might generalize to other consistencies defined by Galois connections.

6. Combinations of Consistencies

In the Cartesian consistencies described thus far, the same over-approximation has been applied to each of the variables in the problem; in other words, each of the preceding Cartesian consistencies was based on a Cartesian domain abstraction where the component abstraction for every variable was the same. As previously mentioned in Section 2.4, there are cases that call for a consistency using different over-approximations for different variables. This chapter deals with consistencies that apply to two such cases: mixed-type consistency, discussed in Section 6.1, and non-uniform consistency, discussed in Section 6.2. Each of these consistencies may be formulated as an $\langle \alpha, \gamma \rangle$ -consistency based on a Cartesian domain abstraction where the component abstractions for the different variables are *different* Galois connections.

The third case calling for complicated consistencies mentioned in Section 2.4, namely consistencies for structured variable types, is dealt with separately in Chapter 8.

6.1 Mixed-Type Consistency

The first case, as suggested in [74], is a *mixed type* CSP, in which the variables are of different types (e.g., real-number variables or set variables). For example, subset bounds consistency was defined in Definition 5.2 for the case where *all* variables are finite set variables; however this consistency is commonly defined as applying to a mixture of finite set variables and integer variables. To distinguish these two definitions, we refer to the latter as *set-integer bounds consistency*, and define it thus:

Definition 6.1: Set-Integer Bounds Consistency [14, 42]

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ and set $\bar{\mathbf{D}} = \bar{\mathcal{D}}_1 \times \dots \times \bar{\mathcal{D}}_n$ of candidate solutions, where for all $i \in [1, n]$ $\bar{\mathcal{D}}_i$ is either a finite subset of $\mathbb{Z}$ (in which case X_i is an integer variable) or a finite subset of the powerset of $\mathbb{Z}$ (in which case X_i is a set variable). Let $C \in \mathbf{C}$ be a constraint.

A domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ is *set-integer bounds consistent* for C if and only if:

- for each variable X_i such that $i \in [1, n]$ and X_i is a set variable we have

$$\bigcap \text{dom}(X_i) = \bigcap \text{dom}'(X_i) \quad \text{and} \quad \bigcup \text{dom}(X_i) = \bigcup \text{dom}'(X_i)$$

- where $\text{dom}'(X_i)$ is the set of all values in $d \in \text{dom}(X_i)$ such that there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ with $d_i = d$, and
- for each variable X_j such that $j \in [1, n]$ and X_j is an integer variable and for each value d such that

$$d \in \{\min \text{dom}(X_j), \max \text{dom}(X_j)\}$$

there exists a valid assignment $\langle d_1, \dots, d_n \rangle \in C$ with $d_j = d$.

An assignment $\langle d_1, \dots, d_n \rangle$ is *valid* if and only if for all $\ell \in [1, n]$ we have that

- $\bigcap \text{dom}(X_\ell) \subseteq d_\ell \subseteq \bigcup \text{dom}(X_\ell)$ if X_ℓ is a set variable, and
- $d_\ell \in [\min \text{dom}(X_\ell), \max \text{dom}(X_\ell)]$ if X_ℓ is an integer variable.

Definition 6.1 could be extended to include more types of variables, by specifying for each new type the same two conditions specified already for set and integer variables: first, the over-approximation of the current domain of each variable, and second, the over-approximation of the set of satisfying tuples. With the exception of bounds(D) consistency and range consistency, these two over-approximations are the same, and therefore induce a Galois connection: these consistencies, in other words, can be characterized by defining one Galois connection for each type of variable.

Set-integer bounds consistency is induced by the Cartesian domain abstraction

$$\wp(\overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n) \xleftrightarrow[\alpha^{\text{B}(\subseteq) + \text{B}(\mathbb{Z})}]{\gamma^{\text{B}(\subseteq) + \text{B}(\mathbb{Z})}} ([\overline{\mathcal{D}}_1] \times \dots \times [\overline{\mathcal{D}}_n]) \quad (6.1)$$

where for all $i \in [1, n]$ the component abstraction for the variable X_i is either

$$\wp(\wp(\mathbb{Z})) \xleftrightarrow[\alpha^{[\wp(\mathbb{Z})]}]{\gamma^{[\wp(\mathbb{Z})]}} [\wp(\mathbb{Z})]$$

from (5.7), or

$$\wp(\mathbb{Z}) \xleftrightarrow[\alpha^{[\mathbb{Z}]}]{\gamma^{[\mathbb{Z}]}} [\mathbb{Z}]$$

from (5.12), depending on whether X_i is a set variable or an integer variable.

That the consistency induced by this Galois connection is the same as subset bounds consistency is a direct consequence of Theorem 5.1:

Proposition 6.2

$\langle \alpha^{\text{B}(\subseteq)}, \gamma^{\text{B}(\subseteq)} \rangle$ -consistency is equivalent to subset bounds consistency.

Proof. Follows directly from Propositions 5.3 and 5.6. □

Section 8.3 presents additional examples of mixed-type consistency, for mixtures of integer variables and string variables (e.g., Section 8.3.1).

6.2 Non-Uniform Consistency

In the second case, namely *non-uniform* consistency, the variables are of the same type, but different over-approximations are used for different variables.

As an example, consider the global constraint $\text{ELEMENT}(X, A_1, \dots, A_k, E)$, which is satisfied by a ground instance in which the element at index X of the sequence $\langle A_1, \dots, A_k \rangle$ must equal E ; in other words, treating $\langle A_1, \dots, A_k \rangle$ as an array $\vec{A}$, ELEMENT enforces the rule $\vec{A}[X] = V$ on ground instances [100]. Relatively little literature exists regarding the propagation of ELEMENT , but it appears that the propagation algorithms used in many CP solvers maintain a consistency weaker than domain consistency [40]. For example, in the CLP(FD) solver of SICStus Prolog [19] the propagator algorithm for ELEMENT maintains what may be described as domain consistency on the index X , and bounds consistency on both the value E and the sequence $\langle A_1, \dots, A_k \rangle$ [18].

Definition 6.3: Element Consistency

Consider a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X, A_1, \dots, A_k, E \rangle$ and set $\bar{\mathbf{D}}$ of candidate solutions. Let $C \in \mathbf{C}$ be the constraint ELEMENT constraint

$$\text{ELEMENT}(X, A_1, \dots, A_k, E)$$

A domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ is *element-consistent* for C if and only if:

- for each value $x \in \text{dom}(X)$ there exists a valid assignment

$$\langle x, a_1, \dots, a_k, e \rangle \in C$$

- for each variable A_j such that $j \in [1, k]$ and each value a_j such that

$$a_j \in \{\min \text{dom}(A_j), \max \text{dom}(A_j)\}$$

there exists a valid assignment

$$\langle x, a_1, \dots, a_k, e \rangle \in C$$

- for each value e such that

$$e \in \{\min \text{dom}(E), \max \text{dom}(E)\}$$

there exists a valid assignment

$$\langle i, v, a_1, \dots, a_k \rangle \in C$$

An assignment $\langle x, a_1, \dots, a_k, e \rangle$ is *valid* if and only if

- $x \in \text{dom}(X)$,
- $a_j \in [\min \text{dom}(A_j), \max \text{dom}(A_j)]$ for all $j \in [1, k]$, and
- $e \in [\min \text{dom}(E), \max \text{dom}(E)]$.

Element consistency is induced by the Cartesian domain abstraction

$$\wp(\overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n) \xleftrightarrow{(\gamma^D \circ (\gamma^{\text{id}} \times \gamma_1^{[\mathbb{Z}]} \times \dots \times \gamma_k^{[\mathbb{Z}]} \times \gamma_{\epsilon}^{[\mathbb{Z}]}) / (\alpha^{\text{id}} \times \alpha_1^{[\mathbb{Z}]} \dots \times \alpha_k^{[\mathbb{Z}]} \times \alpha_{\epsilon}^{[\mathbb{Z}]}) \circ \alpha^D} ([\overline{\mathcal{D}}_1] \times \dots \times [\overline{\mathcal{D}}_n]) \quad (6.2)$$

where α^D is the Cartesian abstraction function (4.2), γ^D is the Cartesian concretization function (4.3), $\alpha^{\text{id}} \circ \gamma^{\text{id}}$ is the identity function on $\mathbb{Z}$, $\alpha_j^{[\mathbb{Z}]}$ for all $j \in [1, k]$ and $\alpha_{\epsilon}^{[\mathbb{Z}]}$ are interval abstraction functions (5.1), and $\gamma_j^{[\mathbb{Z}]}$ for all $j \in [1, k]$ and $\gamma_{\epsilon}^{[\mathbb{Z}]}$ are interval concretization functions (5.2).

That the consistency induced by this Galois connection is the same as element consistency is a direct consequence of Theorem 4.4:

Proposition 6.4

$\langle \alpha^{B(\subseteq)}, \gamma^{B(\subseteq)} \rangle$ -consistency is equivalent to subset bounds consistency.

Proof. Follows directly from Theorem 4.4 and Proposition 5.6. □

Part II:

Constraints on String Variables

The engine can arrange and combine its numerical quantities exactly as if they were *letters* or any other *general* symbols; ... it would be a mistake to suppose that because its *results* are given in the *notation* of a more restricted science, its *processes* are therefore restricted to those of that science.

AUGUSTA ADA KING,
COUNTESS OF LOVELACE

A *string* is a (possibly empty) finite sequence of elements selected from a finite set of symbols, Σ , called an *alphabet*. Constraint problems that include strings as unknowns occur in a wide variety of verification problems, such as test case generation [34], program analysis [16], model checking [39], and web security [15]. CP has a lot of potential for computer-aided verification [23], and has set the state-of-the-art for some such problems, witness [24] for instance. In CP, such an unknown string may be modeled as a *string variable* with a set of strings as its current domain.

String variables are an example of a *structured variable type*. A structured variable represents a discrete-valued object composed of multiple *components* that are mutually related in some fundamental fashion [42].

For example, the best known structured variable type for CP solvers is the *set variable* type, for which the subset interval abstraction, introduced in Section 5.2.1, is a common domain representation [41, 77]. A subset interval-based domain representation has two components: a set that defines the greatest lower bound of the domain, and another that defines the least upper bound of the domain. The concrete domain of sets that is represented by a particular pair of these component sets is determined by a fundamental relationship between them: the concrete domain consists of every set that is both a superset of the greatest lower bound and a subset of the least upper bound. Extensions of the subset interval-based domain representation introduce additional components, notably adding an integer interval representing candidate cardinalities [6, 7] or a pair of lexicographic set bounds [83]. Other representations are possible, most notably using reduced-order binary decision diagrams [57] or length-lex set bounds [103].

Structured objects may appear as features of high-level modeling languages for combinatorial problems, but commonly recurring structured objects may also be implemented in a CP solver as a new variable type, effectively encapsulating the recurring combinatorial substructure of the object in a variable implementation.

A central issue in the implementation of a structured variable type is the space complexity of the corresponding domain representation, which is typically greater than that for a primitive variable type, and is often exponential in the number of components of the structured type. For example, the domain of a string variable may be represented as a set of natural numbers as follows: assume that the alphabet Σ of size n is ordered such that $\Sigma = \langle a_1, \dots, a_n \rangle$, then each string $w = a_{i_k} a_{i_{k-1}} \dots a_{i_1} a_{i_0}$ may be mapped to a natural number by the function $f_n: \Sigma^* \rightarrow \mathbb{N}^+$ where:

$$f_n(w) = i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \dots + i_1 \cdot n + i_0$$

For alphabet Σ , the string w is said to be the *base- n notation* for the natural number $f_n(w)$ [30]. If the domain of a string variable is limited to strings of length b or less, then the size of a base n representation of that domain would be $\mathcal{O}(n^b)$. For this reason, implementations of structured variable types generally

represent and reason on over-approximations rather than exact representations of the variable domains.

In Section 2.3, a consistency corresponding to a choice of domain representation was referred to as a *representational consistency*. The fundamental difference between a computational consistency and a representational one is that an over-approximation of domains corresponding to the latter determines a kind of information granularity: propagator functions cannot reason on the contents of a concrete domain, because its approximation, which corresponds to an abstract domain, is the only information represented in the CP solver. For structured variable types, the representation is atomic: propagator algorithms may only prune variable domains by modifying the components of the variable.

Some aspects of representational consistency have been touched upon in previous sections. Lemma 3.4 states that the smallest sound abstraction of an exact propagator function on the abstract lattice of a representational consistency will always be contracting, and Proposition 3.7 states that the corresponding canonical propagator function is idempotent.

* * *

Part II of this thesis puts the theoretical framework of Part I into practice as a central part of the design, development, and implementation of a string variable type for constraint programming. The development starts from first principles, and is informed throughout by the Galois-based framework of Part I.

Part II proceeds as follows:

- Chapter 7 introduces a general formulation of the string variable type for CP solvers, and constraints thereon.
- Chapter 8 formally describes a bounded-length over-approximation of a set of strings, discusses the application of that approximation as a domain representation for string variables, and derives propagator functions for string constraints from Section 7.2.
- Chapter 9 explores the idea of implementing a structured variable as an aggregation of built-in variable types, and shows how this choice of representation relates to the Galois connection-based framework from Part I.
- Chapter 10 details the design and implementation of string variables as a new variable type for a CP solver.

7. String Variables and Constraints

This chapter presents string variables in the most general sense, as variables with domains that are sets of strings. A practical representation of string variable domains, which will later form the basis of an efficient implementation, is developed in the next chapter.

Section 7.1 defines string variables, irrespective of their actual implementation, Section 7.2 describes constraints over string variables, and Section 7.3 reviews related work on string constraint solving.

Notation for Strings and Languages

An alphabet Σ is a finite set of *symbols*. A string w of length $|w| = n$ over the alphabet Σ is a finite sequence of n elements of Σ called the *characters* of w . For example, the string $x = \text{'aab'}$ is a sequence of three characters from the alphabet $\Sigma = \{\text{'a'}, \text{'b'}\}$, which is a set of two symbols. The i th character of a given string w is denoted by $w[i]$, so $w = w[1]w[2] \cdots w[n]$, where $w[i] \in \Sigma$ for all $i \in [1, n]$ (for all other values of i , $w[i]$ is undefined); continuing the previous example, the characters $x[1]$ and $x[2]$ are both the symbol 'a', and the character $x[3]$ is the symbol 'b'. The unique string of length 0, called the *empty string*, is denoted by ϵ ; thus $\epsilon[i]$ is undefined for all $i \in \mathbb{N}^+$.

Let $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$ be strings over the alphabet Σ . Then x and y are said to be *equal*, written in the usual way as $x = y$, if and only if $n = m$ and $x[i] = y[i]$ for all $i \in [1, n]$. The *concatenation* of x and y , which is written as $x \cdot y$, is the string $x[1] \cdots x[n]y[1] \cdots y[m]$. String y is said to be a *substring* of string x if and only if there exist strings w and z over Σ such that $x = w \cdot y \cdot z$. For $1 \leq i, j \leq |x|$, the substring from the i th to the j th character of x is written as $x[i:j]$; if $i > j$, then $x[i:j] = \epsilon$. The *reverse* of x is the string $x^{\text{rev}} = x[n] \cdots x[1]$.

For $n \in \mathbb{N}$, the set of strings over Σ of length n is denoted by Σ^n , and Σ^0 is defined to be the singleton set containing ϵ . The infinite set of all strings over Σ , including ϵ , is denoted by Σ^* . A possibly infinite set $\mathcal{L} \subseteq \Sigma^*$ of strings is called a *language* over Σ .

If the binary relation $<$ is a partial order on the alphabet Σ , then the natural dictionary ordering of strings in Σ^* , written $<_{\text{lex}}$ and called the *lexicographic* order induced by $<$, is a partial order on Σ^* ; if $<$ is a total order, then $<_{\text{lex}}$ is a total order, as well.

A string homomorphism, or simply a *morphism*, is a function on strings that substitutes a single string for each symbol: if h is a homomorphism on the alphabet Σ and $w = w[1] \cdots w[n]$ is a string over Σ , then $h(w)$ is the string $h(w[1])h(w[2]) \cdots h(w[n])$.

7.1 String Variables

The domain of a string variable is the language of all strings that are either feasible or not yet known to be infeasible; in other words, the domain of a string variable is an element of $\wp(\Sigma^*)$, which is the powerset lattice of Σ^* . We call this lattice the *concrete string lattice* for a single string variable. From the standpoint of implementation, an element of the concrete string lattice is not an ideal representation for the domain of a string variable: operations on languages, and hence propagation of constraints over string variables represented as languages, are computationally expensive. For example, [44] describes a CP solver that uses a deterministic finite automaton to represent the domain of a string variable, which is therefore restricted to being a regular language over Σ^* ; even with this restriction, propagation remains computationally impractical.

In Chapter 8 an over-approximation of string variable domains is presented that does allow for practical propagation algorithms. This over-approximation is then subsequently used as the basis for constructing an efficient string constraint solver.

7.2 String Constraints

In this section several constraints over string variables are defined. The list is by no means exhaustive; other constraints over string variables are certainly conceivable. However, the constraints presented here are sufficient to model a large number of string constraint problems.

The constraints are defined for an arbitrary alphabet Σ . Without loss of generality, the existence of a scalar symbol variable is assumed, i.e., a variable with a domain that is a subset of Σ ; in practice Σ could be mapped to any scalar variable type. Similarly, for the moment no distinction will be drawn between string variables of fixed-length, bounded-length, or unbounded-length; that distinction is of course relevant to any discussion of propagator algorithms, but it is irrelevant for the definition of constraints.

The constraints that follow are categorized either as constraints involving only string variables (Section 7.2.1), constraints that include another variable type (Section 7.2.2), or constraints of language membership (Section 7.2.3).

7.2.1 Pure String Constraints

We refer to constraints involving only string variables as *pure string constraints*.

Definition 7.1: EQ Constraint

The constraint $\text{EQ}(X_1, X_2)$ holds if string variables X_1 and X_2 are equal, where equality for strings means that they have equal length and the same symbol at each index.

The constraint EQ is implemented by the exact propagator function

$$\text{EQ}^E: \wp(\Sigma^* \times \Sigma^*) \rightarrow \wp(\Sigma^* \times \Sigma^*)$$

where

$$\text{EQ}^E(\mathcal{T}) = \{\langle x, y \rangle \in \mathcal{T} \mid x = y\}$$

Definition 7.2: NEQ Constraint

The constraint $\text{NEQ}(X_1, X_2)$ holds if string variables X_1 and X_2 are not equal, where inequality for strings means that the strings have different lengths or that there exists an index for which the two strings have a different symbol.

The constraint NEQ is implemented by the exact propagator function

$$\text{NEQ}^E: \wp(\Sigma^* \times \Sigma^*) \rightarrow \wp(\Sigma^* \times \Sigma^*)$$

where

$$\text{NEQ}^E(\mathcal{T}) = \{\langle x, y \rangle \in \mathcal{T} \mid x \neq y\}$$

Definition 7.3: REV Constraint

The constraint $\text{REV}(X_1, X_2)$ holds if string variable X_1 is equal to the reverse of string variable X_2 ,

The constraint REV is implemented by the exact propagator function

$$\text{REV}^E: \wp(\Sigma^* \times \Sigma^*) \rightarrow \wp(\Sigma^* \times \Sigma^*)$$

where

$$\text{REV}^E(\mathcal{T}) = \{\langle x, y \rangle \in \mathcal{T} \mid x = y^{\text{rev}}\}$$

Definition 7.4: CAT Constraint

The constraint $\text{CAT}(X_1, X_2, X_3)$ holds if the concatenation of string variables X_1 and X_2 is equal to the string variable X_3 .

The constraint CAT is implemented by the exact propagator function

$$\text{CAT}^E: \wp(\Sigma^* \times \Sigma^* \times \Sigma^*) \rightarrow \wp(\Sigma^* \times \Sigma^* \times \Sigma^*)$$

where

$$\text{CAT}^E(\mathcal{T}) = \{\langle x, y, z \rangle \in \mathcal{T} \mid x \cdot y = z\}$$

The lexicographic ordering relationship is useful in the context of strings, suggesting the following constraint.

Definition 7.5: LEX Constraint

The constraint $\text{LEX}(X_1, X_2)$ holds if the string variable X_1 lexicographically precedes string variable X_2 .

The constraint LEX is implemented by the exact propagator function

$$\text{LEX}^E: \wp(\Sigma^* \times \Sigma^*) \rightarrow \wp(\Sigma^* \times \Sigma^*)$$

where

$$\text{LEX}^E(\mathcal{T}) = \{\langle x, y \rangle \in \mathcal{T} \mid x \leq_{\text{lex}} y\}$$

Mapping one string to another via substitution is accomplished by the following constraint, parameterized by a morphism h :

Definition 7.6: MORPH_h Constraint

The constraint $\text{MORPH}_h(X_1, X_2)$ holds if the string variable X_2 is a morph of the string variable X_1 defined by the morphism h .

The constraint MORPH_h is implemented by the exact propagator function

$$\text{MORPH}_h^E: \wp(\Sigma^* \times \Sigma^*) \rightarrow \wp(\Sigma^* \times \Sigma^*)$$

where

$$\text{MORPH}_h^E(\mathcal{T}) = \{\langle x, y \rangle \in \mathcal{T} \mid h(x) = y\}$$

For convenience, Definition 7.6 assumes that the two alphabets of the morphism are the same, but this is not a restriction, as one may always allow Σ to be equal to $\Sigma_1 \cup \Sigma_2$. When discussing propagator algorithms for MORPH_h , it is useful to distinguish whether the morphing h is injective or not, and whether h maps symbols to strings over Σ or only to symbols in Σ ; Definition 7.6 encompasses all of these cases.

7.2.2 Mixed String Constraints

We refer to constraints involving at least one string variable and at least one non-string variable as *mixed string constraints*.

Definition 7.7: LEN Constraint

The constraint $\text{LEN}(X_1, X_2)$ holds if the string variable X_1 has a length equal to the integer variable X_2 .

The constraint LEN is implemented by the exact propagator function

$$\text{LEN}^E: \wp(\Sigma^* \times \mathbb{N}) \rightarrow \wp(\Sigma^* \times \mathbb{N})$$

where

$$\text{LEN}^E(\mathcal{T}) = \{\langle x, \ell \rangle \in \mathcal{T} \mid \ell = |x|\}$$

Definition 7.8: SUB Constraint

The constraint $\text{SUB}(X_1, X_2, X_3)$ holds if the string variable X_2 is a contiguous substring of string variable X_1 , starting at the index given by the integer variable X_3 .

The constraint SUB is implemented by the exact propagator function

$$\text{SUB}^E: \wp(\Sigma^* \times \Sigma^* \times \mathbb{N}^+) \rightarrow \wp(\Sigma^* \times \Sigma^* \times \mathbb{N}^+)$$

where

$$\text{SUB}^E(\mathcal{T}) = \{\langle x, y, i \rangle \in \mathcal{T} \mid y = x[1] \cdots x[i + |y| - 1]\}$$

The special case of SUB in which the substring has a fixed length of one (i.e., where X_2 can be replaced by a scalar variable) is distinguished as the constraint CHAR .

Definition 7.9: CHAR Constraint

The constraint $\text{CHAR}(X_1, X_2, X_3)$ holds if the string variable X_1 has a character at the index given by the integer variable X_3 that is equal to the symbol given by the symbol variable X_2 .

The constraint CHAR is implemented by the exact propagator function

$$\text{CHAR}^E: \wp(\Sigma^* \times \Sigma \times \mathbb{N}^+) \rightarrow \wp(\Sigma^* \times \Sigma \times \mathbb{N}^+)$$

where

$$\text{CHAR}^E(\mathcal{T}) = \{\langle x, c, i \rangle \in \mathcal{T} \mid c = x[i]\}$$

Definition 7.10: ALPHABET Constraint

The constraint $\text{ALPHABET}(X_1, X_2)$ holds if the set variable X_2 is exactly the set of symbols that occur in the string variable X_1 .

The constraint ALPHABET is implemented by the exact propagator function

$$\text{ALPHABET}^E: \wp(\Sigma^* \times \wp(\Sigma)) \rightarrow \wp(\Sigma^* \times \wp(\Sigma))$$

where

$$\text{ALPHABET}^E(\mathcal{T}) = \left\{ \langle x, s \rangle \in \mathcal{T} \mid s = \bigcup_{i=1}^{|x|} \{x[i]\} \right\}$$

7.2.3 Language Membership Constraints

Conceptually, a constraint stating that a string variable X_1 is a member of a formal language $\mathcal{L}$ may be viewed as a unary constraint on X_1 parameterized by $\mathcal{L}$, irrespective of what type of language $\mathcal{L}$ is. In practice, propagation algorithms for language membership constraints are entirely dependent on the class of the language, so it is common to name language membership constraints by the class of the language.

Thus, for a language $\mathcal{L}$ we have the constraint $\text{REGULAR}_{\mathcal{L}}$ if $\mathcal{L}$ is regular, and $\text{CONTEXTFREE}_{\mathcal{L}}$ if $\mathcal{L}$ is context-free.

Definition 7.11: REGULAR $_{\mathcal{L}}$ Constraint

The constraint $\text{REGULAR}_{\mathcal{L}}(X_1)$ holds if the string variable X_1 is a member of the regular language $\mathcal{L}$.

The constraint $\text{REGULAR}_{\mathcal{L}}$ is implemented by the exact propagator function

$$\text{REGULAR}_{\mathcal{L}}^E: \wp(\Sigma^*) \rightarrow \wp(\Sigma^*)$$

where

$$\text{REGULAR}_{\mathcal{L}}^E(\mathcal{T}) = \{ \langle x \rangle \in \mathcal{T} \mid x \in \mathcal{L} \}$$

Definition 7.12: $\text{CONTEXTFREE}_{\mathcal{L}}$ Constraint

The constraint $\text{CONTEXTFREE}_{\mathcal{L}}(X_1)$ holds if the string variable X_1 is a member of the context-free language $\mathcal{L}$.

The constraint $\text{CONTEXTFREE}_{\mathcal{L}}$ is implemented by the exact propagator function

$$\text{CONTEXTFREE}_{\mathcal{L}}^E: \wp(\Sigma^*) \rightarrow \wp(\Sigma^*)$$

where

$$\text{CONTEXTFREE}_{\mathcal{L}}^E(\mathcal{T}) = \{\langle x \rangle \in \mathcal{T} \mid x \in \mathcal{L}\}$$

7.3 Related Work: String Solvers

String constraint solving has been the focus of a large amount of research in recent years. Current string constraint solving methods may be broadly classified according to their treatment of string length. At one extreme are *fixed*-length string solvers, such as [54, 68]. These solvers are typically extensional in the sense that they generate solutions individually. Fixed-length solvers are generally superior to unbounded-length solvers for producing a single solution, but are inferior for producing the set of all solutions. At the opposite end of the spectrum are solvers for string variables of *unbounded* length, such as [1, 36, 39, 44, 51, 52, 65, 97, 106]. These solvers define the set of all satisfying strings intensionally, typically by formal languages. Constraint reasoning in these solvers generally reduces to a question of language intersection; research is focused on avoiding the exponential blowup of these intersection operations.

7.3.1 Fixed-length String Variables

In CP, a fixed-length string variable has a natural representation as an array of scalar variables. Such a representation allows the modeler to take advantage of the large number of global constraints made available by existing CP solvers. Of particular interest for solving string problems are constraints for membership in regular [11, 75] and context-free [78, 90] languages. For example, the propagator in [75] for $\text{REGULAR}_{\mathcal{L}}$ works by maintaining a *layered graph*: the regular language is specified by a finite automaton, and each layer of the graph replicates the nodes of the automaton but with each transition connecting to a node in the next layer. Candidate characters for the corresponding unknown string (in this case represented not by a string variable, but rather as a fixed-length array of integer variables) are determined by the labels of the transitions between nodes in two consecutive layers. Propagation works by removing transitions that are not on a path between the start node in the first layer and any accepting node in the last layer.

Other fixed-length string constraint solvers reason on strings bit-wise, instead of character-wise, encoding a string as a fixed-length bit-vector. For example, HAMPI [38, 54] provides a theory of fixed-length strings for *satisfaction modulo theories* (SMT) solvers, using the bit-vector solver STP [37]. HAMPI handles constraints of membership in both regular and context-free languages. For a set of such constraints, on a *single* fixed-length string variable, HAMPI either returns one satisfying string, or reports that the constraints are unsatisfiable. Fixed-length bit-vector variables have also been explored in CP [68].

7.3.2 Unbounded-length String Variables

Fixed-length string constraint problems are trivially decidable, as every fixed-length string variable by definition has a finite domain. When considering strings of *unbounded* length, however, decidability is no longer a guarantee; indeed, for any reasonably complete set of string constraints, unbounded-length string problems are not decidable in general [16]. Fragments of the logic of string constraints *are* decidable for unbounded-length strings, such as the theory of word equations. A *word equation* [61] is a constraint of equality over concatenations of string variables; for a problem with a set $\mathbf{X}$ of string variables, a word equation is an assertion of the form:¹

$$X_{\alpha_1} \cdot \dots \cdot X_{\alpha_k} = X_{\beta_1} \cdot \dots \cdot X_{\beta_\ell} \text{ where } \forall i \in [1, k], j \in [1, \ell] : X_{\alpha_i}, X_{\beta_j} \in \mathbf{X}$$

In [65], Makanin describes an NP-Hard decidability procedure for Boolean combinations of word equations. The inclusion of even a small set of additional string constraints affects the decidability of the theory of word equations; for example, there is a PSPACE-complete decidability procedure for the theory of word equations with regular language membership [61], while on the other the decidability of the theory of word equations with length constraints remains an open problem [16, 17]. Research on solving string constraints for unbounded-length string variables is focused on identifying decidable fragments of the logic of string constraints that are useful, and designing efficient solution procedures for those fragments.

In recent years, a large number of stand-alone string solvers have appeared, each of which handles a slightly different fragment of the logic of string constraints (generally including word equations with regular language membership). Most of these solvers encode string constraint problems as automata problems. For example, DPRLE [51] is a stand-alone solver for systems of word equations over concatenation and regular membership, where regular languages can be constrained by the subset relation; the decision procedure works by building dependency graphs for the variables and constraints, and then generating automata which encode the complete set of satisfying strings.

¹Strictly speaking, this is a *word equation in the theory of concatenation*; the concept is, however, more commonly referred to in the literature simply as a *word equation*.

STRSOLVE [52] handles automaton intersection operations by lazily constructing cross-products; solving proceeds in two phases, first by eagerly constructing an automaton that encodes a high-level description of the search space, then by searching through a space of mappings from variables to paths through *nondeterministic finite automata* (NFAs) that encode the constraints. A successful search yields a set of automata from which a satisfying assignment to the string constraints is generated. SUSHI [36] handles a restricted fragment of string constraints called *simple linear string equations*, which are equalities in which no string variable appears more than once, and string variables occur only on the left-hand side; SUSHI can handle concatenation, substring, regular membership, and regular replacement. REVENANT [39] uses boolean satisfiability with interpolation to check intersections of regular languages defined as *symbolic automata*, i.e., automata with transitions labelled by formulæ over some theory, instead of with constants.

A general drawback of automaton-based approaches is that there is no natural encoding of length constraints. One approach for handling this limitation has been to solve instead a decomposition of the problem, using one solver for arithmetic constraints and another, automata-based solver for string constraints. For example, [34] combines a linear arithmetic solver with a string solver handling equality, disequality and regular language membership constraints (but not concatenation); the algorithm is approximate and PSPACE-complete. Alternately, length constraints may be encoded into automata; for example, in [104] the possible lengths of strings in a given regular language are encoded in two types of length automata: the first type is combined with other arithmetic automata to allow for complex constraints on string length, while the second type is used to determine upper and lower bounds on possible string lengths. The result, however, is only a weak connection between string lengths and other numerical constraints. Other solvers that combine automata and numerical reasoning [1, 43, 81] are able to strengthen this connection to varying degrees.

Alternately, a decision procedure for a decidable fragment of the logic of string constraints can be used as a theory of strings for a DPLL(T)-based *satisfiability modulo theories* (SMT) solver. The first string theory for an SMT solver was Z3-STR [106], an extension of the SMT solver Z3 that provides a theory of word equations with length constraints. As reported in [60], not only is Z3-STR unsound (i.e., it reports that some satisfiable problems are unsatisfiable), but it also returns spurious solutions to some problems. Both Z3-STR2 [105] and S3 [97] make significant corrections and extensions to Z3-STR's theory of strings, most notably the addition of constraints of regular-language membership (S3 also reports numerous significant improvements to the efficiency of the core algorithm). Unfortunately, while both Z3-STR2 and S3 eliminate the spurious solutions of Z3-STR, S3 at least is demonstrably unsound (see [2]); it is unclear whether Z3-STR2 corrects the unsoundness present in the original. A more recent pair of DPLL(T)-based theorem solvers for

word equations with both length and regular language membership, `NORN` [1, 2] and the string theory extension of `cvc4` [60], appear at this time to be both sound and correct. Termination of the decision procedure used each of these $\text{DPLL}(T)$ -based solvers is guaranteed for some fragment of the theory of word of equations with regular language membership and length constraints; the exact nature of the fragments where known (see [2, 97]) differs among the solvers.

Encoding string constraint problems with symbolic automata (see above) is an approach of particular interest for SMT-based string solving, as the formulæ of the labels provide a link between existing theorem solvers for SMT and the automata that encode the string constraints. For example, `REX` [102] constructs symbolic automata to form acceptors for combinations of regular languages and lengths; a linear arithmetic solver further constrains the length variables. `BEK` [50] translates an imperative language into symbolic finite state transducers, defined in any of the theories understood by the SMT solver, yielding a decidable theory.

There is less research into the use of CP for solving string constraints on unbounded-length strings. An early example of this approach in CP is $\text{CLP}(\Sigma^*)$, an extension of constraint logic programming to regular sets of strings. Constraint logic programming (CLP) combines logic programming, a powerful declarative paradigm in which programs are stated as logical relations, with techniques for solving constraints. $\text{CLP}(\Sigma^*)$ includes `CAT` and `REGULARL` constraints, and guarantees termination even in the presence of infinitely large sets;² the cost, however, is restrictions on both the soundness and the completeness of deduction rules.

Closely related is [44, 45], in which the *regular domain* of a string variable is defined by a regular language. A regular domain is represented as a finite automaton accepting that language, and propagation of a constraint over string variables is achieved by computing a series of automaton operations, such as intersection, cross-product, or negation. The expressivity of regular domains is balanced out by relatively expensive propagation: several of the presented propagators take time quadratic in the size of the involved automata, and the size of the automata themselves may grow exponentially with the number of constraints. It is not surprising that performing propagation on string variables of unbounded length by computing on a set of strings is expensive: the equivalent for the domains of integer variables would be propagation over multiple integer variables through computation on value tuples, which is not generally reasonable. Constraints of regular language membership are, of course, trivially enforceable on the regular domain, although extension to context-free languages is impractical.

²When faced with a string constraint that might generate an infinite number of bindings, $\text{CLP}(\Sigma^*)$ avoids it, charmingly, “thanks to the use of floundering” ([42]).

7.3.3 Bounded-length String Variables

Bounded-length string solvers are more flexible and expressive than fixed-length solvers, while sidestepping the undecidability of problems with unbounded-length strings.

Probably the best known tool for bounded-length strings is `KALUZA` [84], which solves constraints in two stages. In the first, an SMT solver is used to find possible lengths for strings that satisfy a conjunction of explicit length constraints, length constraints implied by the string constraints of the problem, and any other integer constraints present in the model. In the second stage, these lengths are applied to create a fixed-length bit-vector problem, solved with STP in the same manner as that employed by `HAMPI`. If the problem in the second stage is unsatisfiable, the first stage is repeated, with the addition of new constraints to avoid previously tried lengths. Strictly speaking, `KALUZA` is not a string constraint *solver* at all: if the problem is determined to be satisfiable, then a witness is returned for a single string variable, selected by the user in advance, while values for the other variables cannot be obtained from the tool. Thus, answers to string constraint problems generated by `KALUZA` are inherently unverifiable. Furthermore, `KALUZA` is noticeably unstable; the tool generates incorrect results, even on instances of its own benchmark (as also reported in [60, 106]), is frequently error-prone, and seems to be no longer in development.

Another stand-alone bit-vector solver for bounded-length strings is described in [16], which, however, neither handles regular-language membership, nor propagates information from numerical constraints to the string variables.

In CP, an early example of a bounded-length string solver is `CLP(S)` [80], or constraint logic programming (CLP) with strings, in which string variables are sequences with explicitly represented length information. Although an extension of a CLP system, `CLP(S)` handles unknown strings through a partial unification procedure, rather than as part of the constraint solving process. Another approach is the constraint-based string description language reported in [33]. The language, which is implemented in a constraint logic programming language for finite domains, `CLP(FD)`, includes the `LEN`, `MORPHh`, and `SUB` constraints, and also handles both approximate and weighted matches between strings. String variables are represented as sequences of characters up to some maximum length.

More recently, propagation for bounded-length sequences of variables in CP is described in [64], which treats open global constraints. A constraint is *global* if its arity is not determined *a priori*. The arity of a *closed* global constraint is determined by the model, and remains constant throughout the solution procedure; in contrast, the arity of an *open* global constraint is determined as the solution procedure progresses [8]. In [64], the scope of an open global constraint is a sequence of scalar variables with a length that has an upper bound determined by an integer variable. During the solution procedure, scalar vari-

ables are *added* to the end (never the beginning) of the sequence, in connection with changes to the bounds of the length. The dynamic addition of variables to constraint scopes is pre-supposed to be handled by a meta-program, not itself described in [64], that controls the CP solver during the solution process.

Our [IV] introduced a representation for bounded-length string variables based on prefix-suffix pairs, and described propagator functions implementing several string constraints upon that representation. The representation presented in this thesis, used as well in our [VI], is much simpler. It should be noted that the consistencies of propagators acting on these two representations are incomparable; furthermore, the propagator functions in the present work have been derived in a principled fashion, using the theoretical framework of Part I, whereas those of [IV] were *ad hoc*.

There are parallels between the bounded-length string variables presented in this dissertation and the *path variables* of [47]. A path variable represents a sequence of visited nodes in a graph; the length of the path is not known in advance, but rather grows dynamically during the solution process. Thus, path variables are structurally similar to the bounded-length string representation we introduce in Chapter 8. The domain of a path variable consists of a fixed prefix of already visited nodes, and a single set of candidate nodes, representing the possible extensions of the path by one node; propagation is always limited to reasoning on possible successor nodes to the current prefix. Our representation, in contrast, allows propagation to affect characters at *any* index in the string. Path variables also preclude repeated values in the prefix by design, a choice motivated by their intended application, which requires an ALLDIFFERENT constraint on nodes in a path; while a perfectly sensible design decision in the stated context, this seems to have limited application in string constraint contexts. On the other hand, path variables enforce no upper bound on length. However, path variables are not, strictly speaking, unbounded sequences: the representation is linear in the number of nodes in the path, so the finiteness of machine memory imposes a *de facto* upper limit—unlike an unbounded-length string solver, path variables cannot help to prove unsatisfiability. Nevertheless, the quasi-bounded approach taken in [47] could have interesting string applications.

8. Over-Approximating a String Domain

Propagation of string constraints such as those defined in Section 7.2 can be computationally practical with an appropriate choice of string variable representation, i.e., a representation based on a sound over-approximation of a set of strings. We consider here a choice of representation appropriate for strings of *bounded* length. This representation was originally described in [VI], where for historical reasons it was called the *open-sequence representation*, and where for space reasons the theoretical framework that underlies the choice of representation was omitted.

In this chapter, armed with the Galois connection-based framework presented in Part I, we provide a complete theoretical exposition of this bounded-length string representation. The over-approximation itself, and a corresponding abstract lattice, is formally described in Section 8.1. Considerations arising from the use of abstract domains as the basis for variable domain representations are dealt with in Section 8.2. Then, in Section 8.3, the over-approximation formalized in Section 8.1 is used to derive, in a principled fashion, propagator functions implementing the constraints defined in Chapter 7.

8.1 Bounded-Length Sequence Representation

For any finite set $\mathcal{W}$ of strings over an alphabet Σ there exists an upper bound of the lengths of the strings in $\mathcal{W}$, i.e., some number $b \in \mathbb{N}^+$ such that for every string $w \in \mathcal{W}$ the length of w is not greater than b . Such a set $\mathcal{W}$ may be over-approximated by a pair $\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle$, consisting of a sequence $\vec{\mathcal{A}}^b$ of sets $\langle \mathcal{A}[1], \dots, \mathcal{A}[b] \rangle$ over Σ , where every $\mathcal{A}[i]$ is the set of all symbols occurring at index $i \in [1, b]$ for some string in $\mathcal{W}$, and an abstract interval $\mathcal{N}$ of natural numbers (i.e., a pair of ordered elements from $\mathbb{N}$, as defined in Section 5.1), whose corresponding concrete interval (i.e., a contiguous subset of $\mathbb{N}$) represents the lengths of the strings in $\mathcal{W}$, with $\max \gamma^{[\mathbb{N}]}(\mathcal{N}) \leq b$. We call this over-approximation the *bounded-length sequence representation*, and refer to a pair $\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle$ as a *b-length sequence*. A *b-length sequence* $\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle$ may be viewed as the abstract representation of the set of all strings that have a length $\ell \in \gamma^{[\mathbb{N}]}(\mathcal{N})$ and a character at each index $i \in [1, \ell]$, where $\ell \leq b$, taken from the set of symbols $\mathcal{A}[i]$.

Two aspects of the over-approximation of the set of candidate lengths merit clarification: first the over-approximation of the set of candidate lengths by an interval, and second the choice of zero as the lower bound of that set.

Interval Length Approximation

The set of candidate lengths in the bounded-length sequence representation is over-approximated by an interval closure operator, a choice motivated by the belief, born out by our experience [III, IV, VI] implementing propagator algorithms for string constraints, that reasoning on upper and lower bounds of string length yields the most effective propagation. It is nonetheless straightforward to modify the representation, and the propagator functions derived in Section 8.3, to support reasoning directly on the actual set of candidate lengths.

Lower Bound of Lengths

Candidate lengths in the bounded-length sequence representation are taken from the set $\mathbb{N}$ of natural numbers *with zero*. The inclusion of zero allows a b -length sequence representation to represent sets of strings that include the empty string, ϵ , i.e., the unique zero-length string. A conjunction of concatenation constraints is frequently used to model the construction of one unknown string from two or more unknown strings in sequence. As the identity of concatenation is ϵ , a string variable domain representation that cannot represent ϵ is akin to an integer variable domain representation that cannot represent zero: either makes some models harder to express.

Example 8.1

Consider a string constraint problem with unknown strings w , x , y , z , and s , and regular language $\mathcal{L}$, subject to the following conjunction of constraints:

$$w \in \mathcal{L} \wedge x \in \mathcal{L} \wedge w \cdot s = y \wedge x \cdot s = z$$

In other words, each of y and z is the concatenation of some string in $\mathcal{L}$, followed by the same possibly empty suffix s . If string variables are allowed to take ϵ as a value, then this problem may be naturally modeled as a CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_w, X_x, X_y, X_z, X_s \rangle$, set of candidate solutions $\bar{\mathbf{D}} = (\Sigma^*)^5$, and set of constraints:

$$\mathbf{C} = \left\{ \begin{array}{l} \text{REGULAR}_{\mathcal{L}}(X_w), \text{REGULAR}_{\mathcal{L}}(X_x), \\ \text{CAT}(X_w, X_s, X_y), \text{CAT}(X_x, X_s, X_z) \end{array} \right\}$$

If, however, string variables are *not* allowed to take ϵ as a value, then the same problem becomes:

$$(w \in \mathcal{L} \wedge x \in \mathcal{L}) \wedge \left((w \cdot s = y \wedge x \cdot s = z) \vee (w = y \wedge x = z) \right) \quad (8.1)$$

In CP, disjunctive models generally result in poor propagation; as an example, a CSP modeling (8.1) would require a constraint expressing the *negation* of concatenation, which is a very weak constraint indeed.

Galois Connection for Bounded-length Strings

For an alphabet Σ we denote the powerset lattice of strings of length not more than b as

$$\langle \wp(\bigcup_{\ell=0}^b \Sigma^\ell), \subseteq, \cup, \cap \rangle$$

We denote the set of all b -length sequences on the same alphabet as $\mathbb{S}_b$. We call the following function $\gamma^{\mathbb{S}_b}$, which maps a b -length sequence to the corresponding set of strings, the *b -length sequence concretization function*:

$$\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}^b, \mathcal{N}) = \bigcup_{\ell \in \gamma^{[\mathbb{N}]}(\mathcal{N})} \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}[i] \right\} \quad (8.2)$$

We call the inverse mapping $\alpha^{\mathbb{S}_b}$, which maps a set of strings to a b -length sequence, the *b -length sequence abstraction function*:

$$\alpha^{\mathbb{S}_b}(\mathcal{W}) = \left\langle \left\{ \{w[1] \in \Sigma \mid w \in \mathcal{W}\}, \dots, \{w[b] \in \Sigma \mid w \in \mathcal{W}\} \right\}, \alpha^{[\mathbb{N}]}(\{|w| \in \mathbb{N} \mid w \in \mathcal{W}\}) \right\rangle \quad (8.3)$$

These two maps form a Galois connection, which we call the *b -length sequence approximation*:

$$\langle \wp(\bigcup_{\ell=0}^b \Sigma^\ell), \subseteq, \cup, \cap \rangle \xleftrightarrow[\alpha^{\mathbb{S}_b}]{\gamma^{\mathbb{S}_b}} \langle \mathbb{S}_b, \leq^{\mathbb{S}_b}, \wedge^{\mathbb{S}_b}, \vee^{\mathbb{S}_b} \rangle \quad (8.4)$$

It will now be shown that (8.4) is in fact a Galois connection, by constructing it from two other Galois connections. The first of these Galois connections over-approximates a set of strings by considering only string length:

Lemma 8.1

The following is a Galois connection between the powerset lattice of strings of length not more than b and the lattice of natural number intervals:

$$\wp(\bigcup_{\ell=0}^b \Sigma^\ell) \xleftrightarrow[\alpha^n]{\gamma^n} [\mathbb{N}] \quad (8.5)$$

where

$$\alpha^n(\mathcal{W}) = \alpha^{[\mathbb{N}]}(\{|w| \mid w \in \mathcal{W}\}) \quad (8.6)$$

$$\gamma^n(\mathcal{N}) = \left\{ w \in \Sigma^\ell \mid \ell \in \gamma^{[\mathbb{N}]}(\mathcal{N}) \right\} \quad (8.7)$$

Proof. Follows directly from the fact that $\gamma^n \circ \alpha^n$ is a closure operation on $\wp(\bigcup_{\ell=0}^b \Sigma^\ell)$. $\square$

The second Galois connection over-approximates a set of strings by considering only the set of symbols at each index. Before showing that it is, in fact, a Galois connection, we first show that the concretization function is monotonic:

Lemma 8.2

Let $\gamma^a: \wp(\Sigma)^b \rightarrow \wp(\bigcup_{\ell=0}^b \Sigma^\ell)$, where $\wp(\Sigma)^b$ is a poset under the component-wise set inclusion order $\subseteq$ as defined in Section 4.1, be the function:

$$\gamma^a(\mathcal{A}[1], \dots, \mathcal{A}[b]) = \bigcup_{\ell=0}^b \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}[i] \right\}$$

Then γ^a is monotonic.

Proof. Let $\vec{\mathcal{A}}_1^b, \vec{\mathcal{A}}_2^b \in \wp(\Sigma)^b$ be the tuples $\vec{\mathcal{A}}_1^b = \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[b] \rangle$ and $\vec{\mathcal{A}}_2^b = \langle \mathcal{A}_2[1], \dots, \mathcal{A}_2[b] \rangle$, such that $\vec{\mathcal{A}}_1^b \subseteq \vec{\mathcal{A}}_2^b$. By the definition of component-wise subset inclusion, the set $\mathcal{A}_1[i]$ is a subset of the set $\mathcal{A}_2[i]$ for all $i \in [1, b]$. Clearly we have:

$$\left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}_1[i] \right\} \subseteq \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}_2[i] \right\}$$

for all $\ell \in [0, b]$. Therefore, $\gamma^a(\vec{\mathcal{A}}_1^b) \subseteq \gamma^a(\vec{\mathcal{A}}_2^b)$, and γ^a is monotonic. $\square$

From Lemma 8.2 it follows that the following pair of maps between sets of strings and sequences of sets of symbols is a Galois connection:

Lemma 8.3

The following is a Galois connection between the powerset lattice of strings of length not more than b and the tuples of b sets of symbols:

$$\wp\left(\bigcup_{\ell=0}^b \Sigma^\ell\right) \xrightleftharpoons[\alpha^a]{\gamma^a} \left\langle \wp(\Sigma)^b, \subseteq, \cup, \cap \right\rangle \quad (8.8)$$

where $\subseteq$ (respectively, $\cup$, $\cap$) is the component-wise set inclusion (respectively, component-wise set union, component-wise set intersection) relation, as defined in Section 4.1, and where:

$$\alpha^a(\mathcal{W}) = \left\langle \{w[1] \mid w \in \mathcal{W}\}, \dots, \{w[b] \mid w \in \mathcal{W}\} \right\rangle \quad (8.9)$$

$$\gamma^a(\mathcal{A}[1], \dots, \mathcal{A}[b]) = \bigcup_{\ell=0}^b \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}[i] \right\} \quad (8.10)$$

Proof. Let $\vec{\mathcal{A}}^b = \langle \mathcal{A}[1], \dots, \mathcal{A}[b] \rangle$ be a sequence of sets over the alphabet Σ , and $\mathcal{W} \subseteq \wp(\bigcup_{\ell=0}^b \Sigma^\ell)$ be a set of strings over Σ , such that $\alpha^a(\mathcal{W}) \subseteq \vec{\mathcal{A}}^b$. Then

we have:

$$\begin{aligned}
\alpha^a(\mathcal{W}) &\subseteq \vec{\mathcal{A}}^b \\
&\Leftrightarrow \langle \{w[1] \mid w \in \mathcal{W}\}, \dots, \{w[b] \mid w \in \mathcal{W}\} \rangle \subseteq \vec{\mathcal{A}}^b && [\text{by (8.9)}] \\
&\Leftrightarrow \gamma^a(\langle \{w[1] \mid w \in \mathcal{W}\}, \dots, \{w[b] \mid w \in \mathcal{W}\} \rangle) \subseteq \gamma^a(\vec{\mathcal{A}}^b) && [\text{by Lem. 8.2}] \\
&\Leftrightarrow \bigcup_{\ell=0}^b \left\{ w' \in \Sigma^\ell \mid \forall i \in [1, \ell] : w'[i] \in \{w[i] \mid w \in \mathcal{W}\} \right\} \subseteq \gamma^a(\vec{\mathcal{A}}^b) && [\text{by (8.10)}]
\end{aligned} \tag{8.11}$$

Let $z \in \mathcal{W}$ be a string. Clearly $z_i \in \{w_i \mid w \in \mathcal{W}\}$ for all $i \in [1, |z|]$. It follows that:

$$\mathcal{W} \subseteq \bigcup_{\ell=0}^b \left\{ w' \in \Sigma^\ell \mid \forall i \in [1, \ell] : w'[i] \in \{w[i] \mid w \in \mathcal{W}\} \right\}$$

So by (8.11), we have that $\mathcal{W} \subseteq \gamma^a(\vec{\mathcal{A}}^b)$; therefore (8.8) is a Galois connection by Definition 2.8. $\square$

We now demonstrate that (8.4) is in fact a Galois connection:

Proposition 8.4

A b -length sequence approximation (8.4) is a Galois connection.

Proof. For all $\mathcal{W} \in \wp(\bigcup_{\ell=0}^b \Sigma^\ell)$ we have:

$$\begin{aligned}
&\langle \alpha^a(\mathcal{W}), \alpha^n(\mathcal{W}) \rangle \\
&= \left\langle \langle \{w[1] \mid w \in \mathcal{W}\}, \dots, \{w[b] \mid w \in \mathcal{W}\} \rangle, \right. \\
&\quad \left. \alpha^{[\mathbb{N}]} \left(\left\{ |w| \mid w \in \mathcal{W} \right\} \right) \right\rangle && [\text{by (8.6) and (8.9)}] \\
&= \alpha^{\mathbb{S}_b}(\mathcal{W}) && [\text{by (8.3)}]
\end{aligned}$$

Furthermore, for all $\vec{\mathcal{A}}^b \in \wp(\Sigma)^b$ and $\mathcal{N} \in [\mathbb{N}]$ we have:

$$\begin{aligned}
&\gamma^a(\vec{\mathcal{A}}^b) \cap \gamma^n(\mathcal{N}) \\
&= \bigcup_{\ell=0}^b \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}[i] \right\} \cap && [\text{by (8.7) and (8.10)}] \\
&\quad \left\{ w \in \Sigma^\ell \mid \ell \in \gamma^{[\mathbb{N}]}(\mathcal{N}) \right\} \\
&= \bigcup_{\ell \in \gamma^{[\mathbb{N}]}(\mathcal{N})} \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}[i] \right\} \\
&= \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}^b, \mathcal{N}) && [\text{by (8.2)}] \tag{8.12}
\end{aligned}$$

Thus by Lemma 2.13(iii), the direct product of the Galois connections (8.5) and (8.8) is the Galois connection $\wp(\bigcup_{\ell=0}^b \Sigma^\ell) \xleftrightarrow[\alpha^{\mathbb{S}_b}]{\gamma^{\mathbb{S}_b}} \mathbb{S}_b$. $\square$

8.2 Over-Approximations as Domain Representations

In an implementation of a string variable type based on a bounded-length sequence representation, the domain of each string variable X_i is represented by a b -length sequence $\langle \vec{\mathcal{A}}_i^b, \mathcal{N}_i \rangle$. The domain of X_i is the concrete set of strings corresponding to that representation; in other words, we have:

$$\text{dom}(X_i) = \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_i^b, \mathcal{N}_i) \quad (8.13)$$

Thus, since the b -length sequence is the representation of X_i , the domain of X_i is always a set of strings that is representable by a b -length sequence.

When discussing a b -length sequence $\langle \vec{\mathcal{A}}_i^b, \mathcal{N}_i \rangle$ that represents the domain of a string variable X_i , we refer to the set $\mathcal{N}$ of string lengths as the set of *candidate lengths*, and to each set $\mathcal{A}[i]$ of characters as the set of *candidate characters* at index i ; together, the candidate lengths and candidate characters of $\langle \vec{\mathcal{A}}_i^b, \mathcal{N}_i \rangle$ are called its *candidate values*. Collectively, we refer to these sets as the *components* of the domain representation.

8.2.1 Consistency

The closure operation $\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}$ induced by the Galois connection (8.4) defines a mapping from the domain of a string variable, i.e., a set of strings, to the smallest set of strings that both over-approximates the domain and is representable by a b -length sequence. For a CP solver that represents string variables as bounded-length sequences, that Galois connection therefore defines the strongest pruning achievable on string variables by any propagator. Using this Galois connection as the component-wise abstraction for each string variable in a CSP, we can construct a Cartesian domain abstraction that defines a consistency corresponding to this pruning strength.

8.2.2 Representation Invariants

A set of b -length sequences $\mathbb{S}_b$ is different from the sets of abstract domains defined in previous sections in one key manner: it contains many pairs of elements that map to the same concrete set of strings.

Example 8.2

The set $\mathbb{S}_3$ of b -length sequences where $b = 3$ contains the elements

$$t_1 = \langle \langle \{1, 2\}, \{1, 2\}, \emptyset \rangle, [1, 3] \rangle$$

and

$$t_2 = \langle \langle \{1, 2\}, \{1, 2\}, \{1, 2\} \rangle, [1, 2] \rangle$$

Both t_1 and t_2 are mapped by $\gamma^{\mathbb{S}_3}$ to the same concrete set of strings, namely $\mathcal{W} = \{ '11', '12', '21', '22' \}$. Equation (8.3) maps the set $\mathcal{W}$ to the following 3-length sequence:

$$\alpha^{\mathbb{S}_3}(\mathcal{W}) = \langle \langle \{1, 2\}, \{1, 2\}, \emptyset \rangle, [1, 2] \rangle$$

While $\alpha^{\mathbb{S}_3}(\mathcal{W})$ is not equal to either t_1 or t_2 , one does have both that $\alpha^{\mathbb{S}_3}(\mathcal{W}) \leq^{\mathbb{S}_3} t_1$ and that $\alpha^{\mathbb{S}_3}(\mathcal{W}) \leq^{\mathbb{S}_3} t_2$.

In fact, several b -length sequences are mapped by $\gamma^{\mathbb{S}_b}$ onto any given concrete set of strings, while $\alpha^{\mathbb{S}_b}$ maps any concrete set of strings onto the greatest lower bound of these b -length sequences. The function $\gamma^{\mathbb{S}_b}$ is not injective, and the function $\alpha^{\mathbb{S}_b}$ is not surjective. There are in fact three equivalent conditions for any Galois connection $\mathcal{P} \xrightleftharpoons[\alpha]{\gamma} \mathcal{P}^\sharp$: the function α is surjective if and only if the function γ is injective, which furthermore holds if and only if $\alpha \circ \gamma$ is the identity function on $\mathcal{P}^\sharp$; such a Galois connection is called a *Galois insertion*.¹

While our earlier examples have in fact been based on Galois insertions, it is important to note that both $\langle \alpha, \gamma \rangle$ -consistency and $(\gamma \circ \alpha)$ -completeness are defined for Galois connections in general. All of the proofs in Chapters 3 to 6 hold for Galois connections for which the abstract lattice might contain superfluous elements. However, when using the abstract lattice as a basis for a variable domain representation in an implementation, it is desirable that each concrete domain corresponds to only one abstract domain. In part, this condition helps to ensure the correctness of the CP solver, but it can also simplify the implementation. Consider the extreme case in which a singleton domain may have multiple valid representations, with the result that even simple operations, such as equality checking, become computationally expensive.

Fortunately, any Galois connection may be reduced to a Galois insertion, forcing the abstraction function to be surjective by removing superfluous elements of the abstract lattice. Recall from Definition 2.7 that a reduction operator is the dual of a closure operator: an *contracting*, monotonic, and idempotent function on a poset.

¹ Alternately, in algebraic contexts, such a Galois connection is called a *reflection*.

Lemma 8.5: Reduction of Galois Connections [73]

Let

$$\langle \mathcal{P}, \leq, \vee, \wedge \rangle \xleftrightarrow[\alpha]{\gamma} \langle \mathcal{P}^\#, \leq^\#, \vee^\#, \wedge^\# \rangle$$

be a Galois connection, let the reduction operator $\rho : \mathcal{P}^\# \rightarrow \mathcal{P}^\#$ be

$$\rho(p^\#) = \bigwedge^\# \{p'^\# \in \mathcal{P}^\# \mid \gamma(p^\#) = \gamma(p'^\#)\} = (\alpha \circ \gamma)(p^\#) \quad (8.14)$$

and let $\rho[\mathcal{P}^\#] = \{\rho(p^\#) \mid p^\# \in \mathcal{P}^\#\}$ be the image of $\mathcal{P}^\#$ under ρ . Then the poset $\langle \rho[\mathcal{P}^\#], \leq^\# \rangle$ is a complete lattice, and $\mathcal{P} \xleftrightarrow[\alpha]{\gamma} \rho[\mathcal{P}^\#]$ is a Galois insertion.

When the abstract lattice defines the domain representation for a variable, the reduction operator $\alpha \circ \gamma$ defines a *representation invariant* that ensures that representations are minimal. In the abstract string lattice $\mathbb{S}_b$, the reduction operator $\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}$ maps a b -length sequence $\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle$ to a minimal b -length sequence $\langle \vec{\mathcal{A}}'^b, \mathcal{N}' \rangle$, representing the same concrete set of strings, but in which the information encoded by the sequence $\vec{\mathcal{A}}'^b$ is consistent with the information encoded by the length $\mathcal{N}'$. In other words, the function $\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}$ ensures that:

$$\mathcal{A}[i] = \emptyset \Leftrightarrow i > \max \gamma^{[\mathbb{N}]}(\mathcal{N}) \quad (8.15)$$

for all $i \in [1, b]$. For any b -length sequence $\vec{\mathcal{A}}^b$ we denote the index of the last symbol in the longest string as $\hat{\ell}$. If there does not exist an $i \in [1, b]$ such that $\mathcal{A}[i] = \emptyset$, then $\hat{\ell} = \max \gamma^{[\mathbb{N}]}(\mathcal{N})$; otherwise, $\hat{\ell} = \min(\max \gamma^{[\mathbb{N}]}(\mathcal{N}), \min(\{i \in [1, b] \mid \mathcal{A}[i] = \emptyset\}) - 1)$. So we can write:

$$\begin{aligned} \alpha^{\mathbb{S}_b}(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}^b, \mathcal{N})) &= \alpha^{\mathbb{S}_b}\left(\bigcup_{\ell \in \mathcal{N}} \left\{w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w_i \in \mathcal{A}[i]\right\}\right) \\ &= \left\langle \left\langle \mathcal{A}[1], \dots, \mathcal{A}[\hat{\ell}], \underbrace{\emptyset, \dots, \emptyset}_{b-\hat{\ell}} \right\rangle, \mathcal{N} \sqcap [0, \hat{\ell}] \right\rangle \end{aligned} \quad (8.16)$$

Intuitively, if any $\mathcal{A}[i]$ is empty, then $\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}^b, \mathcal{N})$ contains no strings of length i or greater; and upon abstraction back to $\mathbb{S}_b$ via (8.16), $\mathcal{A}[j]$ will be empty for all $j \geq i$. Note that $\hat{\ell}$ could be zero, in which case $[0, \hat{\ell}]$ would be the singleton set $\{0\}$, and $\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}^b, \mathcal{N})$ would contain no string except possibly the empty string.

An implementation, then, may eliminate redundant variable domain representations by enforcing the representation invariant:

$$\text{dom}(X_i) = (\alpha \circ \gamma)(\text{dom}(X_i)) \quad (8.17)$$

An important feature of our framework is that we get this representation invariant essentially for free, directly from the Galois connection. For example, we defined the b -length sequence representation without considering the uniqueness of the resulting domain representations, but the Galois connection (8.4) shows us how to reduce any domain representation to a minimal, consistent, and equivalent representation. In fact, as we will see in the next section, $(\gamma \circ \alpha)$ -canonical propagator functions automatically enforce the representation invariant.

8.3 Propagation for Bounded-Length Strings

It is now possible to derive propagator functions for the string constraints defined in Section 7.2, for a Cartesian domain abstraction for which the component abstraction for each integer variable is the interval approximation (5.1) on $\mathbb{Z}$ and the component abstraction for each string variable is the Galois connection (8.4). The scopes of these constraints differ, each consisting of a different sequence of string and integer variables, so a Cartesian domain abstraction of the corresponding arity and type is specified for each constraint; for each constraint, an $(\gamma \circ \alpha)$ -canonical propagator function is derived, where α and γ are, respectively, the abstraction function and concretization function of the corresponding Cartesian domain abstraction.

As shown by Lemma 3.6, $(\gamma \circ \alpha)$ -canonical propagator functions are always idempotent for the set of $(\gamma \circ \alpha)$ -closed domains. For a given CSP $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, if the domain of every string variable $X_i \in \mathbf{X}$ is implemented by a b -length sequence representation, then by (8.13) the domain $\text{dom}(X_i)$ must be $(\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b})$ -closed. Hence, the derived propagator function in Section 8.3.1, which implements a pure string constraint from Section 7.2.1, is idempotent. The derived propagator function in Section 8.3.2, which implements a mixed string constraint from Section 7.2.2, is idempotent only if the involved variables of non-string types are also guaranteed to be closed under the closure operations assumed for that type—specifically, the interval closure operation corresponding to the variable type.

8.3.1 Propagators for Pure String Constraints

Propagator for the Constraint Eq

We denote by $\alpha^{\mathbb{S}_b \times \mathbb{S}_b}$ and $\gamma^{\mathbb{S}_b \times \mathbb{S}_b}$ the abstraction and concretization functions of the Cartesian domain abstraction:

$$\wp(\Sigma^* \times \Sigma^*) \xrightleftharpoons[\alpha^{\mathbb{S}_b \times \mathbb{S}_b}]{\gamma^{\mathbb{S}_b \times \mathbb{S}_b}} \mathbb{S}_b \times \mathbb{S}_b \quad (8.18)$$

with two component abstractions, each of which is the b -length sequence approximation (8.4); hence, by Definition 4.3 we have:

$$\alpha^{\mathbb{S}_b \times \mathbb{S}_b} = (\alpha^{\mathbb{S}_b} \times \alpha^{\mathbb{S}_b}) \circ \alpha^D \quad \text{and} \quad \gamma^{\mathbb{S}_b \times \mathbb{S}_b} = \gamma^D \circ (\gamma^{\mathbb{S}_b} \times \gamma^{\mathbb{S}_b}) \quad (8.19)$$

We now derive for the constraint $\text{Eq}(X_1, X_2)$ the $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function, which we denote by $\text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}$. We begin by defining, for the exact propagator Eq^E implementing Eq , the smallest sound approximation on the abstract lattice of that Cartesian domain abstraction:

$$\begin{aligned} & \widehat{\text{Eq}}^{\mathbb{S}_b \times \mathbb{S}_b}(\langle \vec{\mathcal{A}}_1^b, \mathcal{N}_1 \rangle, \langle \vec{\mathcal{A}}_2^b, \mathcal{N}_2 \rangle) \\ &= (\alpha^{\mathbb{S}_b \times \mathbb{S}_b} \circ \text{Eq}^E) \left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \times \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2) \right) \quad [\text{by (4.8)}] \\ &= \alpha^{\mathbb{S}_b \times \mathbb{S}_b} \left(\left\{ \langle x, y \rangle \in \left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \times \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2) \right) \mid x = y \right\} \right) \quad [\text{by Def. 7.1}] \\ &= \alpha^{\mathbb{S}_b \times \mathbb{S}_b} \left(\left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \cap \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2) \right)^2 \right) \\ &= ((\alpha^{\mathbb{S}_b} \times \alpha^{\mathbb{S}_b}) \circ \alpha^D) \left(\left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \cap \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2) \right)^2 \right) \quad [\text{by (8.19)}] \\ &= (\alpha^{\mathbb{S}_b} \times \alpha^{\mathbb{S}_b}) \left(\begin{array}{c} \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \cap \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2), \\ \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \cap \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2) \end{array} \right) \quad [\text{by (4.2)}] \\ &= \left\langle \begin{array}{c} \alpha^{\mathbb{S}_b}(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \cap \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2)), \\ \alpha^{\mathbb{S}_b}(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \cap \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2)) \end{array} \right\rangle \quad (8.20) \end{aligned}$$

Now we derive the $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function $\text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}$. By (4.9), we have:

$$\begin{aligned} & \text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}(\mathcal{D}) \\ &= \mathcal{D} \cap (\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \widehat{\text{Eq}}^{\mathbb{S}_b \times \mathbb{S}_b}) \left(\begin{array}{c} \alpha^{\mathbb{S}_b}(\text{dom}(X_1)), \\ \alpha^{\mathbb{S}_b}(\text{dom}(X_2)) \end{array} \right) \quad [\text{by (4.9)}] \\ &= (\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \widehat{\text{Eq}}^{\mathbb{S}_b \times \mathbb{S}_b}) \left(\begin{array}{c} \alpha^{\mathbb{S}_b}(\text{dom}(X_1)), \\ \alpha^{\mathbb{S}_b}(\text{dom}(X_2)) \end{array} \right) \quad [\text{by Lem. 3.4}] \\ &= (\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \widehat{\text{Eq}}^{\mathbb{S}_b \times \mathbb{S}_b}) \left(\begin{array}{c} (\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}_1^b, \mathcal{N}_1), \\ (\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}_2^b, \mathcal{N}_2) \end{array} \right) \quad [\text{by (8.13)}] \quad (8.21) \end{aligned}$$

If the b -length sequence approximation (8.4) were a Galois insertion (i.e., if the function $\gamma^{\mathbb{S}_b}$ of (8.2) were injective), then we would have

$$(\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}^b, \mathcal{N}) = \langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle$$

and (8.21) could be simplified accordingly.

Recalling that $\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}$ is the reduction operator from (8.16), we see that the $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function always maps the b -length sequences of its inputs to their minimal representations before pruning the domain. What is more, the propagator function essentially gets this mapping for free, a result of the fact that $\alpha^{\mathbb{S}_b}$ and $\gamma^{\mathbb{S}_b}$ are quasi-inverses:

$$\begin{aligned}
& \text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}(\mathcal{D}) \\
&= \gamma^{\mathbb{S}_b \times \mathbb{S}_b} \left(\begin{array}{c} \alpha^{\mathbb{S}_b} \left(\begin{array}{c} (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}) \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \\ \cap (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}) \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \end{array} \right), \\ \alpha^{\mathbb{S}_b} \left(\begin{array}{c} (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}) \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \\ \cap (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}) \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \end{array} \right) \end{array} \right) \quad [\text{by (8.20)}] \\
&= \gamma^{\mathbb{S}_b \times \mathbb{S}_b} \left(\begin{array}{c} \alpha^{\mathbb{S}_b} \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \right), \\ \alpha^{\mathbb{S}_b} \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \right) \end{array} \right) \quad [\text{by Lem. 2.9(iii)}] \\
&= (\gamma^{\mathbb{D}} \circ (\gamma^{\mathbb{S}_b} \times \gamma^{\mathbb{S}_b})) \left(\begin{array}{c} \alpha^{\mathbb{S}_b} \left(\begin{array}{c} \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \\ \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \end{array} \right), \\ \alpha^{\mathbb{S}_b} \left(\begin{array}{c} \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \\ \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \end{array} \right) \end{array} \right) \quad [\text{by (8.19)}] \\
&= \gamma^{\mathbb{D}} \left(\begin{array}{c} (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \right), \\ (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \right) \end{array} \right) \\
&= (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \right) \times \quad [\text{by (4.3)}] \\
&\quad (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \right) \cap \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b, \mathcal{N}_2 \right) \right)
\end{aligned}$$

So, for the constraint $\text{Eq}(X_1, X_2)$ we can write

$$\text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}(\mathcal{D}) = \text{dom}'(X_1) \times \text{dom}'(X_2)$$

where:

$$\begin{aligned}
& \text{dom}'(X_1) = \text{dom}'(X_2) = \\
&= (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\begin{array}{c} \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b \right) \cap \gamma^{\mathbb{S}_b} \left(\mathcal{N}_1 \right) \right) \cap \\ \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b \right) \cap \gamma^{\mathbb{S}_b} \left(\mathcal{N}_2 \right) \right) \end{array} \right) \quad [\text{by (8.12)}] \\
&= (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_1^b \right) \cap \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_2^b \right) \cap \gamma^{\mathbb{S}_b} \left(\mathcal{N}_1 \right) \cap \gamma^{\mathbb{S}_b} \left(\mathcal{N}_2 \right) \right)
\end{aligned}$$

$$\begin{aligned}
&= (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\bigcup_{\ell=1}^n \left\{ w \in \Sigma^\ell \mid \begin{array}{l} \forall i \in [1, \ell] : \\ w[i] \in \mathcal{A}_1[i] \cap \mathcal{A}_2[i] \end{array} \right\} \cap \left\{ w' \in \Sigma^\ell \mid \ell \in \mathcal{N}_1 \cap \mathcal{N}_2 \right\} \right) \quad \left[\begin{array}{l} \text{by (8.10),} \\ (8.5) \end{array} \right] \\
&= (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\bigcup_{\ell \in \mathcal{N}_1 \cap \mathcal{N}_2} \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}_1[i] \cap \mathcal{A}_2[i] \right\} \right)
\end{aligned}$$

Let $\langle \vec{\mathcal{A}}'^b, \mathcal{N}' \rangle$ be the b -length sequence defined by:

$$\langle \vec{\mathcal{A}}'^b, \mathcal{N}' \rangle = \left\langle \langle \mathcal{A}_1[1] \cap \mathcal{A}_2[1], \dots, \mathcal{A}_1[b] \cap \mathcal{A}_2[b] \rangle, \mathcal{N}_1 \cap \mathcal{N}_2 \right\rangle \quad (8.22)$$

Then we have

$$\begin{aligned}
&(\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b}) \left(\bigcup_{\ell \in \mathcal{N}_1 \cap \mathcal{N}_2} \left\{ w \in \Sigma^\ell \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}_1[i] \cap \mathcal{A}_2[i] \right\} \right) \\
&= (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}) \left(\vec{\mathcal{A}}'^b, \mathcal{N}' \right) \quad \text{[by (8.16)]} \\
&\quad \quad \quad (8.23)
\end{aligned}$$

which is the reduced b -length sequence:

$$= \gamma^{\mathbb{S}_b} \left(\left\langle \mathcal{A}_1[1] \cap \mathcal{A}_2[1], \dots, \mathcal{A}_1[\hat{\ell}] \cap \mathcal{A}_2[\hat{\ell}], \emptyset, \dots, \emptyset \right\rangle, \mathcal{N}_1 \cap \mathcal{N}_2 \cap [1, \hat{\ell}] \right) \quad (8.24)$$

where $\hat{\ell}$ is the index of the last symbol in the longest string of $\langle \vec{\mathcal{A}}'^b, \mathcal{N}' \rangle$.

Hence, the $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function $\text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}$ enforces the representation invariant defined by the reduction operator derived from the Galois connection (8.18).

8.3.2 Propagators for Mixed String Constraints

Propagator for the Constraint LEN

We denote by $\alpha^{\mathbb{S}_b \times [\mathbb{N}]}$ and $\gamma^{\mathbb{S}_b \times [\mathbb{N}]}$ the abstraction and concretization functions of the Cartesian domain abstraction:

$$\wp(\Sigma^* \times \Sigma^*) \xrightleftharpoons[\alpha^{\mathbb{S}_b \times [\mathbb{N}]}]{\gamma^{\mathbb{S}_b \times [\mathbb{N}]}} \mathbb{S}_b \times [\mathbb{N}] \quad (8.25)$$

with two component abstractions, the first of which is the b -length sequence approximation (8.4), and the second of which is the interval approximation (5.1) on $\mathbb{N}$; hence, by Definition 4.3 we have:

$$\alpha^{\mathbb{S}_b \times [\mathbb{N}]} = (\alpha^{\mathbb{S}_b} \times \alpha^{[\mathbb{N}]}) \circ \alpha^{\text{D}} \quad \text{and} \quad \gamma^{\mathbb{S}_b \times [\mathbb{N}]} = \gamma^{\text{D}} \circ (\gamma^{\mathbb{S}_b} \times \gamma^{[\mathbb{N}]}) \quad (8.26)$$

We now derive for the constraint $\text{LEN}(X_1, X_2)$ the $(\gamma^{\mathbb{S}_b \times [\mathbb{N}]} \circ \alpha^{\mathbb{S}_b \times [\mathbb{N}]})$ -canonical propagator function, which we denote by $\text{LEN}^{\mathbb{S}_b \times [\mathbb{N}]}$. We begin by defining, for the exact propagator LEN^E implementing LEN , the smallest sound approximation on the abstract lattice of that Cartesian domain abstraction:

$$\begin{aligned}
& \widehat{\text{LEN}}^{\mathbb{S}_b \times [\mathbb{Z}]}(\langle \vec{\mathcal{A}}_1^b, \mathcal{N}_1 \rangle, [\ell, u]) \\
&= \left(\alpha^{\mathbb{S}_b \times [\mathbb{N}]} \circ \text{LEN}^E \right) \left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \times \gamma^{[\mathbb{N}]}([\ell, u]) \right) \quad [\text{by (4.8)}] \\
&= \alpha^{\mathbb{S}_b \times [\mathbb{N}]} \left(\left\{ \langle x, n \rangle \in \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \times \gamma^{[\mathbb{N}]}([\ell, u]) \mid |x| = n \right\} \right) \quad [\text{by Def. 7.7}] \\
&= \alpha^{\mathbb{S}_b \times [\mathbb{N}]} \left(\left\{ \left\langle x, n \right\rangle \in \bigcup_{\ell \in \gamma^{[\mathbb{N}]}(\mathcal{N}_1)} \left\{ \begin{array}{c} w \in \Sigma^\ell \\ w[i] \in \mathcal{A}_1[i] \end{array} \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}_1[i] \right\} \mid |x| = n \right\} \right) \quad [\text{by (8.2)}] \\
&= \alpha^{\mathbb{S}_b \times [\mathbb{N}]} \left(\left\{ \left\langle x, n \right\rangle \in \bigcup_{\substack{\ell \in \gamma^{[\mathbb{N}]}(\mathcal{N}_1) \cap \\ \gamma^{[\mathbb{N}]}([\ell, u])}} \left\{ \begin{array}{c} w \in \Sigma^\ell \\ w[i] \in \mathcal{A}_1[i] \end{array} \mid \forall i \in [1, \ell] : w[i] \in \mathcal{A}_1[i] \right\} \right\} \right) \\
&= \alpha^{\mathbb{S}_b \times [\mathbb{N}]} \left(\left\{ \langle x, n \rangle \in \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \sqcap [\ell, u]) \times \gamma^{[\mathbb{N}]}([\ell, u]) \right\} \right) \quad [\text{by (8.2)}] \\
&= \alpha^{\mathbb{S}_b \times [\mathbb{N}]} \left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \sqcap [\ell, u]) \times \gamma^{[\mathbb{N}]}(\mathcal{N}_1) \cap \gamma^{[\mathbb{N}]}([\ell, u]) \right) \\
&= \left((\alpha^{\mathbb{S}_b} \times \alpha^{[\mathbb{N}]}) \circ \alpha^D \right) \left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \sqcap [\ell, u]) \times \gamma^{[\mathbb{N}]}(\mathcal{N}_1) \cap \gamma^{[\mathbb{N}]}([\ell, u]) \right) \quad [\text{by (8.26)}] \\
&= (\alpha^{\mathbb{S}_b} \times \alpha^{[\mathbb{N}]}) \left(\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \sqcap [\ell, u]), \gamma^{[\mathbb{N}]}(\mathcal{N}_1) \cap \gamma^{[\mathbb{N}]}([\ell, u]) \right) \quad [\text{by (4.2)}] \\
&= \left\langle \begin{array}{c} (\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \sqcap [\ell, u]), \\ \alpha^{[\mathbb{N}]}(\gamma^{[\mathbb{N}]}(\mathcal{N}_1) \cap \gamma^{[\mathbb{N}]}([\ell, u])) \end{array} \right\rangle \\
&= \left\langle \begin{array}{c} (\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}_1^b, \mathcal{N}_1 \sqcap [\ell, u]), \\ \mathcal{N}_1 \sqcap [\ell, u] \end{array} \right\rangle \quad (8.27)
\end{aligned}$$

Now we derive the $(\gamma^{\mathbb{S}_b \times [\mathbb{N}]} \circ \alpha^{\mathbb{S}_b \times [\mathbb{N}]})$ -canonical propagator function $\text{LEN}^{\mathbb{S}_b \times [\mathbb{N}]}$. By (4.9), we have:

$$\begin{aligned}
& \text{LEN}^{\mathbb{S}_b \times [\mathbb{N}]}(\mathcal{D}) \\
&= \mathcal{D} \cap \left(\gamma^{\mathbb{S}_b \times [\mathbb{N}]} \circ \widehat{\text{LEN}}^{\mathbb{S}_b \times [\mathbb{N}]} \right) \left(\begin{array}{c} \alpha^{\mathbb{S}_b}(\text{dom}(X_1)), \\ \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right) \quad [\text{by (4.9)}]
\end{aligned}$$

$$= \mathcal{D} \cap \left(\gamma^{\mathbb{S}_b \times [\mathbb{N}]} \circ \widehat{\text{LEN}}^{\mathbb{S}_b \times [\mathbb{N}]} \right) \left(\begin{array}{c} (\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}) (\vec{\mathcal{A}}_1^b, \mathcal{N}_1), \\ \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right) \quad [\text{by (8.13)}] \quad (8.28)$$

Note that, as we have not assumed that the domain of the integer variable X_2 is represented as an interval, we can not eliminate the initial set intersection in (8.28) by taking advantage of Lemma 3.4.

$$\begin{aligned} & \left(\gamma^{\mathbb{S}_b \times [\mathbb{N}]} \circ \widehat{\text{LEN}}^{\mathbb{S}_b \times [\mathbb{N}]} \right) \left(\begin{array}{c} (\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}) (\vec{\mathcal{A}}_1^b, \mathcal{N}_1), \\ \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right) \\ &= \left(\gamma^{\mathbb{S}_b \times [\mathbb{N}]} \circ \widehat{\text{LEN}}^{\mathbb{S}_b \times [\mathbb{N}]} \right) \left(\begin{array}{c} \left\langle \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[\hat{\ell}], \emptyset, \dots, \emptyset \rangle, \right. \\ \left. \mathcal{N}_1 \cap [0, \hat{\ell}] \right\rangle, \\ \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right) \quad [\text{by (8.16)}] \\ &= \gamma^{\mathbb{S}_b \times [\mathbb{N}]} \left(\begin{array}{c} \left(\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b} \right) \left(\begin{array}{c} \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[\hat{\ell}], \emptyset, \dots, \emptyset \rangle, \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right), \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right) \quad [\text{by (8.27)}] \\ &= \left(\gamma^{\text{D}} \circ \left(\gamma^{\mathbb{S}_b} \times \gamma^{[\mathbb{N}]} \right) \right) \left(\begin{array}{c} \left(\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b} \right) \left(\begin{array}{c} \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[\hat{\ell}], \emptyset, \dots, \emptyset \rangle, \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right), \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right) \quad [\text{by (8.26)}] \\ &= \gamma^{\text{D}} \left(\begin{array}{c} \left(\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b} \right) \left(\begin{array}{c} \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[\hat{\ell}], \emptyset, \dots, \emptyset \rangle, \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right), \\ \gamma^{[\mathbb{N}]}(\mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2))) \end{array} \right) \\ &= \gamma^{\text{D}} \left(\begin{array}{c} \gamma^{\mathbb{S}_b} \left(\begin{array}{c} \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[\hat{\ell}], \emptyset, \dots, \emptyset \rangle, \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right), \\ \gamma^{[\mathbb{N}]}(\mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2))) \end{array} \right) \\ &= \gamma^{\mathbb{S}_b} \left(\begin{array}{c} \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[\hat{\ell}], \emptyset, \dots, \emptyset \rangle, \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2)) \end{array} \right) \times \\ & \quad \gamma^{[\mathbb{N}]}(\mathcal{N}_1 \cap [0, \hat{\ell}] \cap \alpha^{[\mathbb{N}]}(\text{dom}(X_2))) \\ &= \gamma^{\mathbb{S}_b} \left(\begin{array}{c} \langle \mathcal{A}_1[1], \dots, \mathcal{A}_1[\hat{\ell}], \emptyset, \dots, \emptyset \rangle, \\ \mathcal{N}_1 \cap [0, \hat{\ell}] \cap [\min \text{dom}(X_2), \max \text{dom}(X_2)] \end{array} \right) \times \\ & \quad \gamma^{[\mathbb{N}]}(\mathcal{N}_1 \cap [0, \hat{\ell}] \cap [\min \text{dom}(X_2), \max \text{dom}(X_2)]) \end{aligned}$$

* * *

These derivations are long and, for the most part, wholly mechanical. Therefore, we include only two full derivations: one for the pure string constraint EQ, and one for the mixed string constraint LEN. The same process may be applied to the remaining constraints given in Sections 7.2.1 and 7.2.2.

In the next chapter, we put these derived propagator functions to work, as the basis for the automatic generation of propagator algorithms for string constraints.

9. Aggregate Implementation of Bounded-length String Variables

In Chapter 10, the bounded-length sequence representation that was described in Section 8.2 will be used as the basis for the implementation of a string variable type for a CP solver. There is, however, another possible method for implementing string variables, suggested by the similarity between sets of candidate values and the domains of scalar variables. Namely, the bounded-length sequence representation of a string variable domain could alternately be represented as an aggregation of two components: an array of scalar variables over the alphabet of the string, and an integer variable for the length of the string. We call this kind of implementation for a structured variable type an *aggregate implementation*.

An aggregate implementation of string variables that is based on the bounded-length sequence representation is explored in this chapter. In Section 9.1, a mapping of the components of the string variable type to scalar variables is described. We further consider the proper handling of both empty sets of candidate characters and the representation invariant. In Section 9.2 a significant advantage of the aggregate implementation is revealed: having translated the bounded-length sequence representation into an aggregate implementation, it is possible to *automatically* generate propagator algorithms that maintain $\langle \alpha^{\mathbb{S}_b}, \gamma^{\mathbb{S}_b} \rangle$ -consistency.

9.1 Aggregate Bounded-Length Strings

Without loss of generality, we focus on strings over the alphabet $\Sigma = \mathbb{N}$ of natural numbers.

Consider a string variable X_i implemented by a b -length sequence representation $\langle \vec{\mathcal{A}}_i^b, \mathcal{N}_i \rangle$, as described in Section 8.2. The domain of X_i , as given by (8.13), is the following set of strings:

$$\text{dom}(X_i) = \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_i^b, \mathcal{N}_i)$$

The idea of an *aggregate* implementation is to instead represent the domain of X_i as the combination of several component variables. A b -length sequence representation $\langle \vec{\mathcal{A}}_i^b, \mathcal{N}_i \rangle$, where $\vec{\mathcal{A}}_i^b = \langle \mathcal{A}_i[1], \dots, \mathcal{A}_i[b] \rangle$, can be represented by an array of integer variables $\langle X_{i_1}, \dots, X_{i_b} \rangle$ corresponding to the sequence

of sets of candidate characters $\vec{\mathcal{A}}_i^b$, and an integer variable X_{i_N} corresponding to the set of candidate lengths $\mathcal{N}_i$. To show that $\langle \vec{\mathcal{A}}_i^b, \mathcal{N}_i \rangle$ is represented by such an aggregation of integer variables, we write:

$$\langle \langle \mathcal{A}_i[1], \dots, \mathcal{A}_i[b] \rangle, \mathcal{N}_i \rangle \simeq \langle \langle \text{dom}(X_{i_1}), \dots, \text{dom}(X_{i_b}) \rangle, \text{dom}(X_{i_N}) \rangle \quad (9.1)$$

However, a naïve translation from sets of candidate values to integer variable domains does not yield the desired result, as shown in the following example:

Example 9.1

Consider a string variable X_i with a domain represented by the b -length sequence representation $\langle \vec{\mathcal{A}}_i^4, \mathcal{N}_i \rangle$, where $b = 4$, such that:

$$\langle \vec{\mathcal{A}}_i^4, \mathcal{N}_i \rangle = \langle \langle \{1, 2\}, \{1, 2\}, \{1, 2\}, \emptyset \rangle, [2, 3] \rangle \quad (9.2)$$

By (8.13), the concrete set of strings that makes up the domain of X_i is:

$$\begin{aligned} \text{dom}(X_i) &= \gamma^{\mathbb{S}_4}(\vec{\mathcal{A}}_i^4, \mathcal{N}_i) \\ &= \bigcup_{\ell \in \gamma^{[\mathbb{N}]}(\mathcal{N}_i)} \left\{ w \in \Sigma^\ell \mid \forall j \in [1, \ell] : w[j] \in \mathcal{A}_i[j] \right\} \\ &= \left\{ \begin{array}{l} '11', '12', '21', '22', \\ '111', '112', '121', '122', \\ '211', '212', '221', '222' \end{array} \right\} \end{aligned}$$

Now consider a naïve aggregate implementation of $\langle \vec{\mathcal{A}}_i^4, \mathcal{N}_i \rangle$, namely the CSP $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$ with variables $\mathbf{X} = \langle X_{i_1}, X_{i_2}, X_{i_3}, X_{i_4}, X_{i_N} \rangle$ and set of candidate solutions $\bar{\mathbf{D}} = \mathbb{N}^5$, where

$$\langle \vec{\mathcal{A}}_i^4, \mathcal{N}_i \rangle \simeq \langle \langle \text{dom}(X_{i_1}), \text{dom}(X_{i_2}), \text{dom}(X_{i_3}), \text{dom}(X_{i_4}) \rangle, \text{dom}(X_{i_N}) \rangle$$

In this representation, the domain $\mathcal{D} \subseteq \bar{\mathbf{D}}$ of the aggregate string variable X_i that corresponds to (9.2) is as follows:

$$\mathcal{D} = \{1, 2\} \times \{1, 2\} \times \{1, 2\} \times \emptyset \times [2, 3] = \emptyset \quad (9.3)$$

Because of the presence of the empty set as the domain of one of the integer variables in (9.3), the domain $\mathcal{D}$ is empty, i.e., none of the strings in $\text{dom}(X_i)$ appears to be a solution to $\langle \mathbf{X}, \bar{\mathbf{D}}, \mathbf{C} \rangle$.

Clearly, for an aggregate implementation to be correct it must properly handle set emptiness in its candidate characters. Additionally, an aggregate implementation has to take into account any representation invariant, as described in Section 8.2.2. These issues are now addressed in turn.

9.1.1 Empty Sets of Candidate Characters

The different semantics of emptiness in the structured representation and the aggregate implementation may be reconciled through the inclusion of a reserved symbol, λ , in the domain of each variable in the aggregate implementation that corresponds to a set of candidate characters. The λ symbol is assigned to each variable that represents an index that does not occur in the string, that is, any index beyond the maximum length of the string. Thus, for any set of candidate characters such that $\mathcal{A}_i[j] = \emptyset$ that is represented in a corresponding aggregate implementation by an integer variable X_{ij} , the following equivalence holds:

$$\mathcal{A}_i[j] = \emptyset \Leftrightarrow \text{dom}(X_{ij}) = \{\lambda\}$$

Alternately, structured variable types may be implemented in aggregate using *option type variables*, when such types are available. An option type variable represents an entity whose existence is unknown in the problem being modeled. Option types have recently been added [67] to the MINIZINC constraint modeling language [72], a mid-level modeling language and toolchain that is interfaced with many underlying CP solvers. The (non)existence of an option type variable is indicated by the variable taking either an ordinary value, in which case the variable does exist, or a special value, denoted in [67] as $\top$, in which case the variable does not exist. An important feature of MINIZINC's implementation of option types is that they are translated away before the model is passed to the solver, so that underlying solvers are not required to support option types natively.

The idea of modeling a bounded-length string variable as a pessimistically large fixed-length sequence of characters and allowing multiple occurrences of a special symbol, sometimes called a *padding symbol*, is not novel (e.g., [48]); indeed, it appears to be a generally acknowledged folk wisdom. With the addition of a λ symbol or option type, the aggregate implementation of a b -length sequence representations may be seen as an implementation of the padding approach, albeit with string length explicitly represented as an integer variable. Indeed, the representation invariant of the b -length sequence representation essentially encodes, via reification, the central idea of the padding model, as will be seen in Section 9.1.2.

The aggregate implementation described in Section 9.2 is not based on MINIZINC, and hence makes use of the λ symbol approach.

9.1.2 Representation Invariant

Enforcement of a representation invariant would presumably be encapsulated by any monolithic variable implementation. For an aggregate implementation, however, the representation invariant must be included in the model as an additional constraint. Intuitively, the representation invariant expresses the

combinatorial substructure of the several variables that make up the aggregate implementation of a single structured variable.

For a string variable represented by the aggregate implementation of a b -length sequence, the representation invariant (8.17) may be expressed using the λ character of Section 9.1.1 as the following conjunction of constraints:

$$\text{STRINV}(X_1, \dots, X_b, X_N) \triangleq \forall i \in [1, b] : X_i = \lambda \Leftrightarrow X_N < i \quad (9.4)$$

Alternately, in the `MINIZINC` language, the representation invariant could be expressed as a constraint predicate using option types:

```
predicate str_inv(array[int] of var opt sym: a, var int: n) =
  forall(i in index_set(a)) (absent(a[i]) <-> n < i);
```

Here, a is an integer-indexed array of the option type variant of a variable type sym , with values representing the alphabet. The `absent` predicate holds if and only if the value of the variable is the special value $\top$ of Section 9.1.1.

9.1.3 Propagator Functions in the Aggregate Implementation

Each aggregation of integer variables in the aggregate implementation represents the same combinatorial structures as a single string variable with a domain given by a b -length sequence representation. So the propagator functions derived in Section 8.3, which implement string constraints for the b -length sequence representation, may easily be adapted to the aggregate implementation.

Consider the string variables X_1 and X_2 , with domains:

$$\begin{aligned} \text{dom}(X_1) &= \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^b, \mathcal{N}_1) \\ \text{dom}(X_2) &= \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^b, \mathcal{N}_2) \end{aligned}$$

The constraint $\text{EQ}(X_1, X_2)$ is implemented by the $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function $\text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}$ derived in Section 8.3.1, which computes:

$$\text{dom}'(X_1) \times \text{dom}'(X_2) = \text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}(\text{dom}(X_1) \times \text{dom}(X_2))$$

such that:

$$\text{dom}'(X_1) = \text{dom}'(X_2) = (\gamma^{\mathbb{S}_b} \circ \alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}'^b, \mathcal{N}') \quad (9.5)$$

where $\langle \vec{\mathcal{A}}'^b, \mathcal{N}' \rangle$ is the b -length sequence defined by:

$$\langle \vec{\mathcal{A}}'^b, \mathcal{N}' \rangle = \left\langle \langle \mathcal{A}_1[1] \cap \mathcal{A}_2[1], \dots, \mathcal{A}_1[b] \cap \mathcal{A}_2[b] \rangle, \mathcal{N}_1 \cap \mathcal{N}_2 \right\rangle \quad (9.6)$$

By (8.13), the new domains of X_1 and X_2 also have b -length sequence representations, i.e., we have:

$$\begin{aligned} \text{dom}'(X_1) &= \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1'^b, \mathcal{N}'_1) \\ \text{dom}'(X_2) &= \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2'^b, \mathcal{N}'_2) \end{aligned}$$

for some b -length sequences $\langle \vec{\mathcal{A}}_1^{rb}, \mathcal{N}'_1 \rangle$ and $\langle \vec{\mathcal{A}}_2^{rb}, \mathcal{N}'_2 \rangle$. Rewriting (9.5) gives:

$$\gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_1^{rb}, \mathcal{N}'_1) = \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_2^{rb}, \mathcal{N}'_2) = \gamma^{\mathbb{S}_b}((\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}^{rb}, \mathcal{N}')) \quad (9.7)$$

As discussed in Section 8.2.2, many b -length sequences share the same concretizations; however, it is sufficient to note in this case that (9.7) is satisfied by the obvious choice:

$$\langle \vec{\mathcal{A}}_1^{rb}, \mathcal{N}'_1 \rangle = \langle \vec{\mathcal{A}}_2^{rb}, \mathcal{N}'_2 \rangle = (\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b})(\vec{\mathcal{A}}^{rb}, \mathcal{N}') \quad (9.8)$$

The three b -length sequences in (9.8) and the two representing the domains of X_1 and X_2 can also be translated into the aggregate implementation via (9.1), yielding:

$$\begin{aligned} \langle \vec{\mathcal{A}}_1^b, \mathcal{N}_1 \rangle &\simeq \langle \langle \text{dom}(X_{1_1}), \dots, \text{dom}(X_{1_b}) \rangle, \text{dom}(X_{1_N}) \rangle \\ \langle \vec{\mathcal{A}}_2^b, \mathcal{N}_2 \rangle &\simeq \langle \langle \text{dom}(X_{2_1}), \dots, \text{dom}(X_{2_b}) \rangle, \text{dom}(X_{2_N}) \rangle \\ \langle \vec{\mathcal{A}}_1^{rb}, \mathcal{N}'_1 \rangle &\simeq \langle \langle \text{dom}'(X_{1_1}), \dots, \text{dom}'(X_{1_b}) \rangle, \text{dom}'(X_{1_N}) \rangle \\ \langle \vec{\mathcal{A}}_2^{rb}, \mathcal{N}'_2 \rangle &\simeq \langle \langle \text{dom}'(X_{2_1}), \dots, \text{dom}'(X_{2_b}) \rangle, \text{dom}'(X_{2_N}) \rangle \\ \langle \vec{\mathcal{A}}^{rb}, \mathcal{N}' \rangle &\simeq \left\langle \begin{array}{c} \langle (\text{dom}(X_{1_1}) \cap \text{dom}(X_{2_1})), \dots, \\ (\text{dom}(X_{1_b}) \cap \text{dom}(X_{2_b})) \rangle, \\ (\text{dom}(X_{1_N}) \cap \text{dom}(X_{2_N}) \cap [1, \hat{\ell}]) \end{array} \right\rangle \end{aligned}$$

Recall from Section 8.2.2 that the function $\alpha^{\mathbb{S}_b} \circ \gamma^{\mathbb{S}_b}$ defines the representation invariant for the b -length sequence representation, which can be expressed in the aggregate implementation by the constraint **STRINV** from Section 9.1.2. Letting **STRINV**^A be a propagator function implementing **STRINV**, we see that (9.8) may be rewritten for the aggregate implementation as:

$$\begin{aligned} &\langle \langle \text{dom}'(X_{1_1}), \dots, \text{dom}'(X_{1_b}) \rangle, \text{dom}'(X_{1_N}) \rangle \\ &= \langle \langle \text{dom}'(X_{2_1}), \dots, \text{dom}'(X_{2_b}) \rangle, \text{dom}'(X_{2_N}) \rangle \\ &= \text{STRINV}^A \left(\langle (\text{dom}(X_{1_1}) \cap \text{dom}(X_{2_1})), \dots, \right. \\ &\quad \left. (\text{dom}(X_{1_b}) \cap \text{dom}(X_{2_b})) \rangle, (\text{dom}(X_{1_N}) \cap \text{dom}(X_{2_N}) \cap [1, \hat{\ell}]) \right) \end{aligned}$$

In other words, on the aggregate implementation, propagation of the constraint $\text{Eq}(X_1, X_2)$ is equivalent to component-wise equality over the two aggregations of integer variables, followed by enforcement of the string representation invariant, exactly as one would expect.

Figure 9.1. Revisiting Figure 1.1: a propagator algorithm that maintains a particular consistency may be created via one of two pathways: either a propagator function is derived from the constraint semantics and then translated into an indexical description, or the constraint semantics is translated into the indexical language and then used to synthesize the indexical description. Either way, automated code generation starting from the resulting indexical description yields a solver-specific propagator algorithm.

9.2 Automatic Generation of Propagator Algorithms

The derived $(\gamma \circ \alpha)$ -canonical propagator functions seen thus far have the following interesting property: if one assumes that the input to such a function is a Cartesian product of variable domains, then the function can be translated into a conjunction of restriction operations on variable domains. When applied to a variable domain, such an operation is called an *indexical* [19, 22, 91, 101]: it is an expression of the form $X \triangleleft \sigma$ that restricts the domain of the variable X to its intersection with the set σ :¹

$$X \triangleleft \sigma \triangleq \text{dom}'(X) := \text{dom}(X) \cap \sigma \quad (9.9)$$

An *indexical language* is a high-level solver-independent language for writing a propagator description with indexicals. The extended indexical language of [69] includes arrays and n -ary operations, and its implementation includes the following two automated transformations, illustrated by Figure 1.1 (reprinted here as Figure 9.1):

- A solver-independent *synthesis* of an indexical description of a propagator from a ground checker of its constraint [71].
- A solver-specific *code generation* of a propagator algorithm from an indexical description thereof [69].

Following our ideas in [III, VI], applied there to aggregate implementations of string variable types, we use this system to automatically generate prototype implementations of derived propagator functions.

¹In the literature, σ , which is frequently called a *range expression*, is a subset of the set of candidate values for the variable X , whose construction is subject to certain monotonicity rules, as explained in [19, 22, 101].

In Section 9.2.1, we demonstrate how synthesis applies to an aggregate encoding of a structured variable type, using the derived representation invariant to capture the encoding’s structural information. In Section 9.2.2, we show how derivation, which applies to *any* $\langle \alpha, \gamma \rangle$ -consistency, may be used in place of the automated synthesis described in [71], which is limited to domain or bounds reasoning. The translation is straightforward and could, in principle, be automated using a system such as SPECWARE, a category-theoretic, general-purpose synthesis tool [92]. The propagator description yielded by either synthesis or derivation may then be used to generate a solver-specific propagator algorithm, as described in [69].

9.2.1 Synthesis

We illustrate the synthesis process on the string variable constraints EQ and LEN from Section 7.2, using the aggregate encoding specified in Section 9.1. We assume the alphabet is a subset of $\mathbb{N}$, with zero as the λ symbol, as described in Section 9.1.1.

Checker

As illustrated in Figure 1.1, the input for the synthesis of a propagator description is a checker, expressed in the checker sub-language of the extended indexical language, for the constraint that the propagator description will implement. A *checker* is a logical relation on ground instances, which distinguishes for a constraint those candidate solutions that satisfy the constraint.

Recall that the constraint $\text{EQ}(X_1, X_2)$ holds if string variables X_1 and X_2 are equal, where equality for strings means that they have equal length and the same symbol at each index; expressed as a logical relation on ground instances of string variables, a checker for EQ is $X_1 = X_2$. Applying the definition of string equality yields the equivalent checker:

$$\text{EQ}(X_1, X_2) : |X_1| = |X_2| \wedge \forall i \in [1, |X_1|] : X_1[i] = X_2[i] \quad (9.10)$$

Similarly, the constraint $\text{LEN}(X_1, X_2)$ holds if the string variable X_1 has a length equal to the integer variable X_2 , and has a corresponding checker $|X_1| = X_2$.

Figure 9.2 shows the result of manually translating the checkers for the constraints EQ and LEN. A complete description of the syntax of the extended indexical language may be found in [69], and is not repeated here. For our purposes, it suffices to illuminate a few less obvious features of the syntax. The string X_1 is represented by two components: an integer variable for the length (`vint xn_1`) and an array of integer variables for the string itself (`vint[] x_1`). The checker for EQ is expressed as a logical conjunction of a set of constraints on ground instances of the elements of the aggregate encoding; in this case, one constraint is on the lengths of the two strings (line 3), and

```

1 def EQ(vint[] X_1, vint Xn_1, vint[] X_2, vint Xn_2){
2   checker{
3     (val(Xn_1) == val(Xn_2)) and
4     and(i in (min(rng(X_1)) .. ((min(rng(X_1))+val(Xn_1))-1)))
5       (val(X_1[i]) == val(X_2[i]))
6   }
7 }
8 def LEN(vint[] X_1, vint Xn_1, vint X_2){
9   checker{
10    val(Xn_1) == val(X_2)
11  }
12 }

```

Figure 9.2. Checkers for the string equality constraint $\text{EQ}(\vec{X}_1, X_1^N, \vec{X}_2, X_2^N)$ and string length constraint $\text{LEN}(\vec{X}_1, X_1^N, X_2)$, using the aggregate encoding of Section 9.1, expressed in the checker sub-language of the extended indexical language from [69].

another constraint is on the contents of the arrays (lines 4 and 5). The latter is expressed as an n -ary conjunction of equality constraints, corresponding to the universal quantification in (9.10).

From these checkers, automatic synthesis [71] yields the indexical propagator descriptions in Figure 9.3.

Refinement

Compared with the propagator function $\text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}$ in Section 8.3, we note that while the synthesized propagator correctly filters values in the arrays x_1 and x_2 at indices below the current minimum length, it misses some propagation on xn_1 and xn_2 . The discrepancy arises from the fact that the propagator function was derived using the Galois connection (8.4), which encapsulates information about the structure of string variables. The synthesis process, on the other hand, has no access to such structural information.

This discrepancy may be partly rectified by synthesizing an additional indexical propagator description from the aggregate encoded checker for the representation invariant (9.4). The translation of this checker, and the indexical propagator description automatically synthesized from it, are shown in Figure 9.4.

The greatest common fixpoint of the propagator descriptions STRINVARIANT from Figure 9.4 and LEN from Figure 9.3 is the same as the greatest common fixpoint of the derived propagator function $\text{LEN}^{\mathbb{S}_b \times [N]}$; however, the conjunction of EQ from Figure 9.3 and STRINVARIANT differs from the derived propagator function $\text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}$. Both are demonstrated by Figure 9.5, which shows an indexical propagator description for EQUAL , obtained by the manual refinement of the indexical propagator description from Figure 9.3. The first difference between these propagator descriptions affects indices between the minimum and maximum string lengths, for which $\text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}$ prunes more than EQUAL from Figure 9.3: intuitively, if the intersection of the domains of $x_1[i]$ and $x_2[i]$

```

1 def EQ(vint[] X_1, vint Xn_1, vint[] X_2, vint Xn_2){
2   propagator(gen)::DR{
3     Xn_1 in dom(Xn_2);
4     Xn_2 in dom(Xn_1);
5     forall(i in
6       (min(rng(X_1)) .. ((min(rng(X_1))+min(Xn_1))-1))) {
7       X_1[i] in dom(X_2[i]);
8       X_2[i] in dom(X_1[i]);
9     }
10  }
11 }
12 def LEN(vint[] X_1, vint Xn_1, vint X_2){
13   propagator(gen)::DR{
14     X_2 in dom(Xn_1);
15     Xn_1 in dom(X_2);
16   }
17 }

```

Figure 9.3. Indexical propagator descriptions for $\text{Eq}(\vec{X}_1, X_1^N, \vec{X}_2, X_2^N)$ and $\text{LEN}(\vec{X}_1, X_1^N, X_2)$, obtained via synthesis from the checkers of Fig. 9.2.

```

1 def STRINVARIANT(vint[] X, vint Xn){
2   checker{
3     and(i in rng(X)) (val(X[i]) == 0 <-> val(Xn)<i)
4   }
5   propagator(gen)::DR{
6     forall(i in rng(X)){
7       max(Xn) < i -> X[i] in {0};
8       i <= min(Xn) -> X[i] in (U minus {0});
9       0 == val(X[i]) -> Xn in inf .. (i-1);
10      not 0 memberof dom(X[i]) -> Xn in i .. sup;
11    }
12  }
13 }

```

Figure 9.4. Indexical propagator description of the representation invariant (9.4) for the aggregate encoding of the string variable type, obtained via synthesis from the checker starting on line 2.

```

1 def EQ(vint[] X_1, vint Xn_1, vint[] X_2, vint Xn_2){
2   propagator(improved)::BR{
3     Xn_1 in (min(Xn_2) .. max(Xn_2));
4     Xn_2 in (min(Xn_1) .. max(Xn_1));
5     forall(i in min(rng(X_1) .. (min(rng(X_1)+min(Xn_1))-1))) {
6       X_1[i] in dom(X_2[i]);
7       X_2[i] in dom(X_1[i]);
8     }
9     forall(i in
10      min(rng(X_1)+min(Xn_1) .. min(rng(X_1))+max(Xn_1))) {
11      (dom(X_1[i]) inter dom(X_2[i])) == {0}
12      -> Xn_1 in inf .. (i-1);
13    }
14    post(STRINVARIANT(X_1,Xn_1));
15    post(STRINVARIANT(X_2,Xn_2));
16  }
17 }

```

Figure 9.5. Manual improvement of the indexical description for EQ from Fig. 9.3. The improved propagator description enforces the representation invariant, and replaces domain reasoning on the length variables with bounds reasoning.

is empty, then only those strings in $\text{dom}(X_1)$ and $\text{dom}(X_2)$ that are shorter than i can possibly satisfy the constraint. This additional reasoning is expressed in Figure 9.5 with the `forall` construct on line 10. The second difference arises from the fact that the propagator function $\text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}$ over-approximates string lengths as intervals, while the indexical propagator description for EQ in Figure 9.3 considers all domain values of x_{n_1} and x_{n_2} . The indexical propagator description in Figure 9.5 has been manually modified to reason on length bounds, replacing the domain intersection operations on the length variables x_{n_1} and x_{n_2} with interval intersection operations (lines 3–4).

The indexical propagator description for EQ from Figure 9.5 additionally enforces the representation invariant via the `post` construction (lines 14–15). The greatest fixpoint of the resulting propagator description is identical to the result of $\text{Eq}^{\mathbb{S}_b \times \mathbb{S}_b}$; however, the latter is idempotent, while the former is not, a difference discussed further in Section 9.2.2, where it is demonstrated how an indexical propagator description for EQ may alternatively be obtained via translation.

9.2.2 Translation

In cases where automated synthesis does not yield a propagator description of the desired consistency, a propagator description may instead be obtained by first deriving an appropriate propagator function, and then translating the derived propagator function into an equivalent indexical propagator description. We now argue that such a translation is possible in the general case for Cartesian consistencies (i.e., those consistencies implied by domain consistency).

Let C be a constraint on variables $\mathbf{X} = \langle X_1, \dots, X_n \rangle$ with set $\overline{\mathbf{D}} = \overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n$ of candidate solutions, and $\wp(\overline{\mathbf{D}}) \xrightarrow[\alpha]{\gamma} \overline{\mathbf{D}}^\#$ be the following Cartesian domain abstraction:

$$\begin{aligned} \wp(\overline{\mathcal{D}}_1 \times \dots \times \overline{\mathcal{D}}_n) &\xrightarrow[\alpha^D]{\gamma^D} (\wp(\overline{\mathcal{D}}_1) \times \dots \times \wp(\overline{\mathcal{D}}_n)) \\ &\xrightarrow[\alpha_1 \times \dots \times \alpha_n]{\gamma_1 \times \dots \times \gamma_n} (\overline{\mathcal{D}}_1^\# \times \dots \times \overline{\mathcal{D}}_n^\#) \end{aligned}$$

The $(\gamma \circ \alpha)$ -canonical propagator function $C^\# : \overline{\mathbf{D}} \rightarrow \overline{\mathbf{D}}$ for C , given by (4.9), is:

$$C^\#(\mathcal{D}) = \mathcal{D} \cap (\gamma \circ \widehat{C}^\#)(\alpha_1(\text{dom}(X_1)), \dots, \alpha_n(\text{dom}(X_n))) \quad (9.11)$$

where $\widehat{C}^\#$ is the smallest sound abstraction on the abstract lattice $\overline{\mathbf{D}}^\#$ of the exact propagator function C^E implementing C . After applying $\widehat{C}^\#$ and the abstraction and concretization functions of the component abstractions, (9.11) will yield:

$$C^\#(\mathcal{D}) = \mathcal{D} \cap \gamma^D(\mathcal{D}_1, \dots, \mathcal{D}_n) \quad (9.12)$$

for some $\mathcal{D}_1, \dots, \mathcal{D}_n$ such that $\mathcal{D}_i \subseteq \overline{\mathcal{D}}_i$ for all $i \in [1, n]$.

Recall from (4.3) that the Cartesian concretization function γ^D maps a tuple of sets $\langle \mathcal{A}_1, \dots, \mathcal{A}_n \rangle$ to the set of tuples $\mathcal{A}_1 \times \dots \times \mathcal{A}_n$. If we assume that every domain $\mathcal{D} \subseteq \overline{\mathbf{D}}$ is closed under the Cartesian closure operator (i.e., that for all $\mathcal{D} \subseteq \overline{\mathbf{D}}$ we have $\mathcal{D} = \text{dom}(X_1) \times \dots \times \text{dom}(X_n)$), then the right-hand side of (9.12) is the intersection of two Cartesian products, and the intersection operation may be distributed:

$$C^\#(\mathcal{D}) = (\text{dom}(X_1) \cap \mathcal{D}_1) \times \dots \times (\text{dom}(X_n) \cap \mathcal{D}_n) \quad (9.13)$$

For all $i \in [1, n]$, the expression $\text{dom}(X_i) \cap \mathcal{D}_i$ is a contracting operation on $\text{dom}(X_i)$ restricting $\text{dom}(X_i)$ to the set $\mathcal{D}_i$. In other words, we can rewrite (9.13) as a conjunction of indexical expressions:

$$C^\#(\mathcal{D}) \equiv \forall i \in [1, n] : X_i \triangleleft \mathcal{D}_i$$

Thus, any $(\gamma \circ \alpha)$ -canonical propagator function may be transformed into an expression that has a direct translation into the extended indexical language of [69].

Figure 9.6 shows the result of applying this translation individually to the $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function $\text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}$ and to the $(\gamma^{\mathbb{S}_b \times [\mathbb{N}]} \circ \alpha^{\mathbb{S}_b \times [\mathbb{N}]})$ -canonical propagator function $\text{LEN}^{\mathbb{S}_b \times [\mathbb{N}]}$.

While the $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function $\text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}$ is idempotent, as explained in Section 8.3, the indexical propagator description EQ

```

1 def EQ(vint[] X_1, vint Xn_1, vint[] X_2, vint Xn_2){
2   propagator(bls){
3     int L =
4       min(i in rng(X_1))((dom(X_1[i]) inter dom(X_2[i]))=={0})-1;
5     Xn_1 in (min(Xn_2) .. max(Xn_2));
6     Xn_1 in (0 .. L);
7     Xn_2 in (min(Xn_1) .. max(Xn_1));
8     Xn_2 in (0 .. L);
9     forall(i in ((min(rng(X_1)) .. L))){
10      X_1[i] in dom(X_2[i]);
11      X_2[i] in dom(X_1[i]);
12    }
13    forall(i in ((min(rng(X_1))+L)-1) .. max(rng(X_1))))
14      {X_1[i] in {0};}
15    forall(i in ((min(rng(X_2))+L)-1) .. max(rng(X_2))))
16      {X_2[i] in {0};}
17  }
18 }
19 def LEN(vint[] X_1, vint Xn_1, vint X_2){
20   propagator(bls){
21     int L = (min(i in rng(X_1))(val(X_1[i])==0))-1;
22     forall(i in L .. max(rng(X_1))){X_1[i] in {0};}
23     Xn_1 in 0 .. L;
24     Xn_1 in min(X_2) .. max(X_2);
25     X_2 in 0 .. L;
26     X_2 in min(Xn_1) .. max(Xn_1);
27   }
28 }

```

Figure 9.6. Indexical propagator descriptions for $\text{Eq}(\vec{X}_1, X_1^N, \vec{X}_2, X_2^N)$ and $\text{Len}(\vec{X}_1, X_1^N, X_2)$, obtained by translation from the corresponding propagator functions.

shown in Figure 9.6 is not idempotent. This is because EQ is a *sequence*, not a conjunction, of indexical expressions. The fixpoint of EQ , however, is the same as the fixpoint of $\text{EQ}^{\mathbb{S}_b \times \mathbb{S}_b}$ (modulo translation into the aggregate implementation).

If, however, an idempotent propagator *algorithm* for EQ is desired, then it may optionally be obtained automatically via solver-specific *code generation*, as described in [70]. Code generation falls outside the scope of this thesis; in brief, however, idempotency may be guaranteed by generating a propagator algorithm that repeats the propagation until a fixpoint is reached.

* * *

Many structured variable types are amenable to implementation as aggregates of existing variable types, in some cases as a prototyping step before the implementation of a new variable type. For example, in early versions of CP(Graph), an extension of the Gecode CP library [96] providing a graph variable type and constraints thereon, the domains of graph variables were implemented as pairs of set variables corresponding to each graph's set of edges and set of nodes [32]. Indeed, any structured variable type whose instances may be correctly over-approximated by an element of an abstract lattice constructed from lattices corresponding to already available CP solver variable types (e.g., integer variables or set variables) is a candidate for aggregate implementation. A propagator function derived via a Galois connection between the concrete lattice of structured variable domains and the abstract lattice of their over-approximations will then have a direct translation into a propagator function for the aggregate implementation.

10. A Bounded-length String Variable Type

In this chapter, we describe the design and implementation of a bounded-length string variable type for a CP solver. In Section 10.1 we describe the variable type and motivate the design choices of subsequent sections. In Section 10.2 we define restriction operations, which allow propagator algorithms to modify the domain of such a string variable. In Section 10.3 we consider several representations for each of the components of the b -length sequence, and motivate why particular choices have been made. In Section 10.4 we describe string-variable specific systems of propagation events, which describe changes to the domain of a variable.

The design choices described in this chapter are then justified by an experimental evaluation in Chapter 11.

10.1 Overview of the Variable Design

The central decision in implementing a variable type is the choice of *data structure* used to represent a variable domain. Figure 10.1 shows a high-level view of the kind of data structure required to represent a b -length sequence, the domain representation for string variables introduced in Chapter 8. The sets $\vec{\mathcal{A}}_j^b = \langle \mathcal{A}_j[1], \dots, \mathcal{A}_j[b] \rangle$ of candidate characters in the b -length sequence for string variable X_j are stored as some sort of list structure, such as an array or a linked list; for simplicity, this list is illustrated in Figure 10.1 as an array, but other implementations are considered in Section 10.3. The representation of each set $\mathcal{A}_j[i]$ of candidate characters at index i is not specified, but is assumed to be the same for all $i \in [1, b]$. The set of candidate lengths $\mathcal{N}_j$ must be stored as well; in the illustration, $\mathcal{N}_j$ is represented as an interval of indices in $\vec{\mathcal{A}}_j^b$. The set of candidate lengths divides $\vec{\mathcal{A}}_j^b$ into three regions: the *mandatory* candidate characters, with an index less than or equal to $\min(\mathcal{N}_j)$; the *optional* candidate characters, with an index greater than $\min(\mathcal{N}_j)$ but less than or equal to $\max(\mathcal{N}_j)$; and the *forbidden* candidate characters, with an index greater than $\max(\mathcal{N}_j)$. Recall from Section 8.2.2 that the representation invariant for b -length sequences requires that the forbidden candidate characters are exactly those for which the set of candidate characters is empty.

This high-level view leaves many questions unanswered. How are the candidate characters represented? Or the candidate lengths? Is the list of candidate characters allocated all at once, or only as needed? The answers to these questions are context-dependent.

Figure 10.1. High-level view of a data structure for the domain, represented by a b -length sequence with maximum length $b = 15$, of a string variable. Taking as an example string variable X_i (top), the sequence $\tilde{\mathcal{A}}_i^b$ of sets of candidate characters is illustrated as a contiguous array of 15 cells, with indices indicated above; the set $\mathcal{N}_i$ of candidate lengths is the interval $[5, 10]$, the lower bound and upper bound of which are shown by $[$ and $]$ brackets, respectively. The bounds of the set of candidate lengths partition the sets of candidate characters into mandatory, optional, and forbidden regions, as indicated in the figure. For string variable X_j (bottom), the set $\mathcal{N}_j$ of candidate lengths is the singleton $\{7\}$, so the length of X_j is fixed; in this case, there are no optional characters, only mandatory and forbidden ones.

No data structure exists in a vacuum, and every choice of data structure represents a tradeoff between memory requirements and computational complexity. Evaluating the choices made in designing a data structure, therefore, requires considering the context in which the data structure will function.

For a variable implementation, that means considering how the variable implementation interacts with the CP solver, and most importantly how the variable implementation interacts with propagator algorithms. A simplistic view of this interaction is shown in Figure 10.2.

Recall from Chapter 1 that a *propagator algorithm* is here a software entity written for a specific CP solver, encapsulating a procedural inference algorithm that removes candidate values from a domain. We contrast propagator algorithms with the related concept of *propagator functions*: a propagator function, such as seen in Chapter 8, is a contracting mathematical function that has one domain as input, and another domain as output.

In an implementation, generating a copy of the domain with every iteration of a propagator algorithm would be far too memory intensive. Rather, propagator algorithms work by modifying the domains of variables. These domains are not exposed directly to propagator algorithms; rather the variable implementation encapsulates the domain representation and exposes some *restriction operations* by which propagator algorithms can request that the domain be modified. Restriction operations provide an interface to the domain that

Figure 10.2. Interactions of a variable implementation and a propagator algorithm: a propagator algorithm requests changes to the domain of a variable via a *restriction operation*, and a variable implementation notifies a propagator algorithm that its domain has changed via a *propagation event*.

allows the removal of candidate values; no other modifications are allowed, as a propagator algorithm is required to be contracting on domains. A restriction operation is itself a contracting function on the domain of a single variable; as such, restriction operations are closely related to both indexicals (see Section 9.2.2) and propagator functions.

The interaction between variable implementations and propagators is not one-way. When the domain of a variable implementation changes under some propagation algorithm, any other propagator algorithm that implements a constraint on the corresponding variable must be notified, as it is possible that the latter propagator algorithm is no longer at fixpoint. Most CP solvers make use of more fine-grained information than simply which variable has changed. This more detailed information is called a *propagation event* [88]; for an integer variable X , for example, a propagation event might indicate that the upper bound of $\text{dom}(X)$ has changed, or that $\text{dom}(X)$ has become a single value.

The interaction of variable implementation and propagator algorithm in Figure 10.2 is highly simplified. Actual CP solvers maintain a queue of propagator algorithms that are not known to be at a fixpoint; a propagation event alerts the solver that any propagator algorithm that has subscribed to that event is no longer known to be at fixpoint, and should therefore be added to the queue if not already present.

Thus, there are three intimately related decisions to be considered when designing a new variable implementation:

- the *data structure*, which stores the domain representation,
- the *restriction operations*, which define how propagator algorithms may modify domains, and
- the *propagation events*, which communicate changes in a domain back to the propagator algorithms.

A string variable implementation should support restriction operations and propagation events that are useful to propagation algorithms for the constraints

defined in Section 7.2. We now consider an example of such a propagator algorithm for the constraint CAT.

10.1.1 Motivating Example: Concatenation

The $(\gamma^{\mathbb{S}_b \times \mathbb{S}_b \times \mathbb{S}_b} \circ \alpha^{\mathbb{S}_b \times \mathbb{S}_b \times \mathbb{S}_b})$ -canonical propagator function implementing the constraint CAT(X_1, X_2, X_3) for domains represented by b -length sequences (see Chapter 8) can be expressed as the following update rule:

$$\text{CAT} \left(\langle \vec{\mathcal{A}}_1^b, \mathcal{N}_1 \rangle, \langle \vec{\mathcal{A}}_2^b, \mathcal{N}_2 \rangle, \langle \vec{\mathcal{A}}_3^b, \mathcal{N}_3 \rangle \right) = \left\langle \langle \vec{\mathcal{A}}_1^{rb}, \mathcal{N}'_1 \rangle, \langle \vec{\mathcal{A}}_2^{rb}, \mathcal{N}'_2 \rangle, \langle \vec{\mathcal{A}}_3^{rb}, \mathcal{N}'_3 \rangle \right\rangle$$

where the results are defined via update rules on the components of the b -length sequences, first the sets of candidate lengths:

$$\mathcal{N}'_1 := \mathcal{N}_1 \cap [\min(\mathcal{N}_3) - \max(\mathcal{N}_2), \max(\mathcal{N}_3) - \min(\mathcal{N}_2)] \quad (10.1a)$$

$$\mathcal{N}'_2 := \mathcal{N}_2 \cap [\min(\mathcal{N}_3) - \max(\mathcal{N}_1), \max(\mathcal{N}_3) - \min(\mathcal{N}_1)] \quad (10.1b)$$

$$\mathcal{N}'_3 := \mathcal{N}_3 \cap [\min(\mathcal{N}_1) + \min(\mathcal{N}_2), \max(\mathcal{N}_1) + \max(\mathcal{N}_2)] \quad (10.1c)$$

and second the sets of candidate characters:

$$\begin{aligned} \mathcal{A}'_1[i] &:= \begin{cases} \mathcal{A}_1[i] \cap \mathcal{A}_3[i] & \text{if } i \leq \max(\mathcal{N}'_1) \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{A}'_2[i] &:= \begin{cases} \mathcal{A}_2[i] \cap \bigcup_{j \in \mathcal{N}_1 \cap [1, \max(\mathcal{N}'_3) - i]} \mathcal{A}_3[i + j] & \text{if } i \leq \max(\mathcal{N}'_2) \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{A}'_3[i] &:= \begin{cases} \mathcal{A}_3[i] \cap \mathcal{A}_1[i] & \text{if } i \leq \min(\mathcal{N}'_1) \\ \mathcal{A}_3[i] \cap \left(\mathcal{A}_1[i] \cup \bigcup_{j \in \mathcal{N}'_1 \cap [i - \max(\mathcal{N}'_2), i]} \mathcal{A}_2[i - j] \right) & \text{if } \min(\mathcal{N}'_1) < i \leq \max(\mathcal{N}'_1) \\ \mathcal{A}_3[i] \cap \bigcup_{j \in \mathcal{N}'_1 \cap [i - \max(\mathcal{N}'_2), i]} \mathcal{A}_2[i - j] & \text{if } \max(\mathcal{N}'_1) < i \leq \max(\mathcal{N}'_3) \\ \emptyset & \text{if } \max(\mathcal{N}'_3) < i \end{cases} \end{aligned}$$

We now consider the inference performed by a propagator algorithm implementing these update rules, in three stages, considering, first, simple length arithmetic with regard to the sets of candidate lengths; second, possible alignments of the three strings with regard to the sets of candidate characters; and third, possible alignments with regard to the sets of candidate lengths. As a running example, we consider three b -length sequences $\langle \vec{\mathcal{A}}_1^b, \mathcal{N}_1 \rangle$, $\langle \vec{\mathcal{A}}_2^b, \mathcal{N}_2 \rangle$, and $\langle \vec{\mathcal{A}}_3^b, \mathcal{N}_3 \rangle$, with maximum length $b = 15$, and sets of candidate lengths $\mathcal{N}_1 = [4, 8]$, $\mathcal{N}_2 = [3, 7]$, and $\mathcal{N}_3 = [6, 15]$, illustrated as:

Length Arithmetic

The simplest inference performed by our propagator algorithm is the arithmetic adjustment of the candidate lengths, as given by (10.1a) to (10.1c). For example, by (10.1c) the length of X_3 must be at least $\min(\mathcal{N}_1) + \min(\mathcal{N}_2)$:

Hence the candidate length 6, which is an element of $\mathcal{N}_3$, is not an element of $\mathcal{N}'_3$. Also, the length of X_3 cannot be more than $\max(\mathcal{N}_1) + \max(\mathcal{N}_2)$:

Computation of new lengths for X_1 and X_2 is similar.

Recall that the b -length sequence representation over-approximates a set of candidate lengths by an interval. If the set of candidate lengths instead has an explicit representation, then the propagator algorithm may infer the following new set of candidate lengths:

$$\mathcal{N}'_3 \leftarrow \{n \in \mathcal{N}_3 \mid \exists n_1 \in \mathcal{N}_1, n_2 \in \mathcal{N}_2 : n = n_1 + n_2\}$$

This computation is called the *Minkowski sum* of the sets $\mathcal{N}_1$ and $\mathcal{N}_2$. The update rule (10.1c) computes the Minkowski sum of two integer intervals, which collapses to the pair of computations, on the upper and lower bound, described above.

This arithmetic length inference is idempotent, and has a time complexity dominated by the computation of the Minkowski sum on the chosen implementation of the candidate lengths.

Candidate Characters

The update rule given above for a set of candidate characters in $\vec{\mathcal{A}}_1^b$ is straightforward: for strings x , y , and z such that $x \cdot y = z$ we must have $x[i] = z[i]$ for

all $i \in [1, |x|]$. But the update rules for sets of candidate characters in $\vec{\mathcal{A}}_2^{tb}$ and $\vec{\mathcal{A}}_3^{tb}$ are more complicated, as every length in the set of candidate lengths for X_1 represents a different alignment of X_2 and X_3 in the resulting concatenation.

Continuing our running example, consider the set $\mathcal{A}'_3[6]$ of candidate characters at index 6 of X_3 . There are four candidate lengths for X_1 , namely $\{4, 5, 6, 7\}$. If, on the one hand, the length of X_1 were fixed to either of the two smallest of these candidates, then the set of candidate characters $\mathcal{A}'_3[6]$ (indicated below by darker shading) would have to be equal to some specific set of candidate characters in $\vec{\mathcal{A}}_2^{tb}$ (indicated below by lighter shading), as for length 4 we would have:

while for length 5 we would instead have:

If, on the other hand, the length of X_1 were fixed to either of the two largest of these candidates, then in either case the set of candidate characters $\mathcal{A}'_3[6]$ would have to be equal to the same set of candidate characters in $\vec{\mathcal{A}}_1^{tb}$, as for length 6 we would have:

while for length 7 we would instead have:

Thus, we can express the update rule for $\mathcal{A}'_3[6]$ in terms of these three other sets of candidate characters $\mathcal{A}'_1[6]$, $\mathcal{A}'_2[1]$, and $\mathcal{A}'_2[2]$ (indicated in light grey):

$$\mathcal{A}'_3[6] := \mathcal{A}'_3[6] \cup \mathcal{A}'_1[6] \cup \mathcal{A}'_2[1] \cup \mathcal{A}'_2[2]$$

This update rule is a specialization of the second case of the update rule for $\mathcal{A}'_3[i]$. Similar inference applies to the third case of that update rule, as well as to the update rule for $\mathcal{A}'_2[i]$.

The time complexity of these candidate character inferences depends on the computation of set union and intersection on the chosen implementation of the sets of candidate characters; the number of set operations required is $\mathcal{O}(\max(\mathcal{N}_3) \cdot |\mathcal{N}_1|)$. The conjunction of the update rules for candidate characters is idempotent.

Possible Alignments

The final type of inference that can be used in the propagator algorithm is to update the set $\mathcal{N}_1$ of candidate lengths by considering the intersections of sets of candidate characters in $\tilde{\mathcal{A}}_2^b$ and $\tilde{\mathcal{A}}_3^b$. For example, if $\mathcal{N}_1$ were $\{4\}$, then we would have:

If the intersection of $\mathcal{A}_2[2]$ (lightly shaded) and $\mathcal{A}_3[6]$ (darkly shaded) were empty, then none of the candidate solutions would satisfy the constraint: there would be no strings $x \in \text{dom}(X_1)$, $y \in \text{dom}(X_2)$, $z \in \text{dom}(X_3)$ such that $x \cdot y = z$, as for every choice of y and z we would have $y[2] \neq z[6]$. Therefore, 4 can be removed from the set $\mathcal{N}_1$ of candidate lengths for X_1 .

This inference is valid only when considering mandatory characters in X_2 . Observe:

If $\mathcal{A}_2[5] \cap \mathcal{A}_3[9]$ is the empty set, then there may be candidate solutions satisfying the constraint such that the length of X_1 is equal to 4, so long as the length of X_2 in those solutions is at most 4.

If the candidate lengths are represented by intervals, then naturally this inference can only be applied at the lower bound and upper bound of $\mathcal{N}_1$, as intermediate candidate lengths cannot be removed from the interval. It is worth noting, however, that similar reasoning can be performed for intermediate candidate lengths should the chosen representation allow it.

The time required to update the sets of candidate lengths based on possible alignments depends on the implementation of the sets of candidate characters. For explicitly represented sets of candidate lengths, the number of set operations is $\mathcal{O}(\max(\mathcal{N}_3) \cdot \min(\mathcal{N}_2))$. For interval representations of the candidate lengths, only $\mathcal{O}(\min(\mathcal{N}_2))$ set operations are required for a single application of all these update rules; however, the new lower bound of $\mathcal{N}_1$ can then

be checked as well, so idempotency of the conjunction of these update rules would require $\mathcal{O}(\min(\mathcal{N}_2) \cdot |\mathcal{N}_1|)$ set operations.

10.1.2 Solver-based Requirements

In an effort to make this presentation of string variables as widely applicable as possible, we have been purposefully general when discussing CP solvers; for example, the search procedure `search` in Algorithm 1.1 is general enough to apply to the majority of solvers. Up until this point, this high-level view of solvers has been sufficient. In motivating the design choices for the implementation of a new variable type, however, it is no longer possible to be entirely general, as several of those choices will depend, at the very least, upon the general design of the target solver.

The target solver for the implementation of the string variable type introduced in this chapter is the `GEODE` toolkit [96], a mature CP solver with state-of-the-art performance and many deployments in industry. The `GEODE` toolkit, written in `c++`, is both modular and extensible. Significantly for the purposes of this chapter, new variable types implemented in `GEODE` can be directly integrated into the solver kernel, and thus incur no efficiency penalty with respect to built-in types.

As described in Chapter 1, the search procedure implemented by `search` is a backtracking search: when the fixpoint algorithm `propagate` results in a failed domain, the decision made at some previous branch point of the search is undone, effectively returning the search to a previously visited node. The tree is grown from this node, if possible, by choosing another of the mutually exclusive decisions that were determined at that node according to the employed branching heuristic.

CP solvers employ one of two main techniques for restoring a previously visited node during backtracking [86]. The most common technique is *trailing*, in which the changes to the domains of the variables during search are stored on a stack; when backtracking occurs, the stack provides sufficient information that a previously visited node can be restored. A less common technique, employed in `GEODE`, is *copying*, in which the domains (along with other elements of the search state) are copied before a decision is applied; thus, every newly added node of the search tree begins with a fresh copy.¹

The fact that `GEODE` is a copying solver has two main impacts on the design decisions of this chapter. First, in a copying solver the search procedure and the fixpoint algorithm are orthogonal, as opposed to trailing solvers for which all the solver components are affected by backtracking. Thus, fortunately, the

¹To be precise, `GEODE` implements backtracking as copying with *recomputation*: full copies of nodes are created only at some interval of search tree depth, and when an uncopied node is needed it may be recomputed by applying all subsequent branching decisions to the closest copied node on the path to the root of the search tree [85, 96].

choices in this chapter can be made without reference to the details of the search procedure. Second, for a copying system, memory management is critical. The amount of memory required by a data structure used to represent the domain of a variable (as well as data structures used to store the states of other solver components) should therefore be minimal; for GECODE, in particular, memory management is most efficient if these data structures do not increase in size during search [96].

10.1.3 Domain-based Requirements

String length is a natural number, and any finite alphabet has a mapping to the natural numbers (imposing an ordering over the alphabet); therefore the possible implementations for both the set $\mathcal{N}_j$ of candidate lengths and the sets $\mathcal{A}_j[i]$ of candidate characters at all indices i are the same. However, sets of candidate lengths and sets of candidate characters have significantly different characteristics, and hence benefit from different choices of representations.

Length

Regarding the set $\mathcal{N}_j \subset \mathbb{N}$ of candidate lengths, the central question is whether to explicitly represent all elements of the set or to instead over-approximate it. As exemplified in Section 10.1.1, the values of upper and lower bounds of $\mathcal{N}_j$ are more useful during propagation than the interior values of $\mathcal{N}_j$. Furthermore, the *lower* bound of the set of candidate lengths is slightly more interesting than the *upper* bound. Indeed, each index that is less than the lower bound of the set of candidate lengths is *mandatory* in the sense that *every* candidate solution must have some symbol at that index. Hence, the lower bound of the set of candidate lengths defines the smallest part of the string that an implementation should represent explicitly. In contrast, the upper bound of the set of candidate lengths defines the set of indices that are *optional*, in that they participate in only some of the candidate solutions; reasonable implementations might explicitly represent all, some, or none of the sets of candidate characters at each of these indices.

Characters

The exact composition of an alphabet is problem dependent; however, the following properties appear to apply generally to alphabets for string problems:

- An alphabet is typically composed of a manageable number of symbols. This is certainly true for many interesting classes of string constraint problems. For example, in natural-language problems strings are constructed from human-readable alphabets;² the same applies to software

²Of course, not all written languages are alphabetic; however, logographic and ideographic writing systems have already been effectively excluded from consideration here by the definition of a string as a sequence of symbols.

analysis problems such as static analysis or verification. To take another example, alphabets common to computational biology also have a limited number of symbols, based on for example the number of bases (4), codons (64), or amino acids (20).

- In many cases the symbols of an alphabet have an explicit total ordering, in which case the strings of any language on that alphabet have a corresponding lexicographic order. Even in cases where there exists no meaningful order for the alphabet, imposing such an order may prove useful; for example, lexicographic ordering has wide-spread application in symmetry-breaking techniques.
- Intervals on an alphabet, while conceptually well-defined for any totally-ordered alphabet, are not particularly interesting: in contrast to numeric types, the ordering of symbols in an alphabet is arbitrary, in that it does not imply any logical relationship between consecutive symbols. As such, a set of symbols that happens to form an interval is no more likely to be meaningful than any other set of symbols taken from the alphabet.
- As exemplified by Section 10.1.1, the propagation of constraints over string variables with bounded-length sequence domain representations depends heavily on standard set operations (i.e., union and intersection) between sets of candidate characters.

10.2 Restriction Operations

Before choosing a data structure to represent the domain for a string variable type, it is helpful to consider the ways in which propagator algorithms acting on those variables will need to modify their domains. As seen in Section 10.1, propagator algorithms do not directly interact with the data structure; rather the variable implementation encapsulates the domain representation and exposes some restriction operations by which propagator algorithms can request that the domain be modified.

The choice of restriction operations for a variable implementation is motivated primarily by two concerns: each operation should be *useful*, insofar it is required by some propagator algorithm, and it should be *efficient* to execute on the chosen data structure. In Section 10.2.1, we describe restriction operations that are useful for string variable implementations in general. In Section 10.2.2 we focus on those restriction operations that are most useful for string variables with domains represented by bounded-length sequences. The desire to efficiently implement these restriction operations will then help to motivate the design of the data structure implementing the domain, in Section 10.3. Before proceeding, though, in Section 10.2.3 we delve into an interesting question regarding the semantics of string restriction operations, determining which of several possible semantics of string equality and disequality is most suitable in the context of string restriction operations.

Table 10.1. *A selection of restriction operations for string variables.*

restriction		operator
string	equality	$\text{seq}(\text{dom}(X), s) = \{x \in \text{dom}(X) \mid x = s\}$
	disequality	$\text{snq}(\text{dom}(X), s) = \{x \in \text{dom}(X) \mid x \neq s\}$
	$\leq_{\text{lex}}$	$\text{sle}(\text{dom}(X), s) = \{x \in \text{dom}(X) \mid x \leq_{\text{lex}} s\}$
	$>_{\text{lex}}$	$\text{sgr}(\text{dom}(X), s) = \{x \in \text{dom}(X) \mid x >_{\text{lex}} s\}$
	intersection	$\text{sin}(\text{dom}(X), \mathcal{S}) = \{x \in \text{dom}(X) \mid x \in \mathcal{S}\}$
	subtraction	$\text{smi}(\text{dom}(X), \mathcal{S}) = \{x \in \text{dom}(X) \mid x \notin \mathcal{S}\}$
length	equality	$\text{leq}(\text{dom}(X), \ell) = \{x \in \text{dom}(X) \mid x = \ell\}$
	disequality	$\text{lnq}(\text{dom}(X), \ell) = \{x \in \text{dom}(X) \mid x \neq \ell\}$
	$\leq$	$\text{lle}(\text{dom}(X), \ell) = \{x \in \text{dom}(X) \mid x \leq \ell\}$
	$>$	$\text{lgr}(\text{dom}(X), \ell) = \{x \in \text{dom}(X) \mid x > \ell\}$
	intersection	$\text{lin}(\text{dom}(X), \mathcal{L}) = \{x \in \text{dom}(X) \mid x \in \mathcal{L}\}$
	subtraction	$\text{lmi}(\text{dom}(X), \mathcal{L}) = \{x \in \text{dom}(X) \mid x \notin \mathcal{L}\}$
character	equality	$\text{ceq}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] = c\}$
	disequality	$\text{cnq}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] \neq c\}$
	$\leq$	$\text{cle}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] \leq c\}$
	$>$	$\text{cgr}(\text{dom}(X), i, c) = \{x \in \text{dom}(X) \mid x[i] > c\}$
	intersection	$\text{cin}(\text{dom}(X), i, \mathcal{C}) = \{x \in \text{dom}(X) \mid x[i] \in \mathcal{C}\}$
	subtraction	$\text{cmi}(\text{dom}(X), i, \mathcal{C}) = \{x \in \text{dom}(X) \mid x[i] \notin \mathcal{C}\}$

10.2.1 Restriction Operations on String Variables

Table 10.1 lists several restriction operations that might be provided by a string variable implementation. The operations are divided into three categories, affecting strings, lengths, or characters.

The first category of operations works on sets of strings, and consists of three pairs of complementary restriction operations: the equality and disequality operations, which restrict the domain of the variable to be a singleton (seq) or to exclude a single string (snq); the inequality operations, which tighten the lower (sgr) or upper (sle) lexicographic bound of the domain; and the set intersection and set subtraction operations, which restrict the domain to be its intersection with a given set of strings (sin) or to exclude a given set of strings (smi).

The second category of operations works on sets of candidate lengths, and consists of three similar pairs of operations: the equality and disequality operations, which restrict the domain of the variable to strings of a given length (leq) or to exclude all strings of a given length (lnq); the inequality operations, which tighten the lower (lgr) or upper (lle) bound of the set of candidate lengths; and the set intersection and set subtraction operations, which restrict the candidate lengths to their intersection with a given set of integers (lin) or to exclude a given set of integers (lmi).

The third category of operations works on sets of candidate characters, and follows the same pattern: the equality and disequality operations, which restrict the domain of the variable to strings with a given symbol at the specified index (`ceq`) or to exclude all strings with a given symbol at the specified index (`cnq`); the inequality operations, which tighten the lower (`cgr`) or upper (`cle`) bound of the set of candidate characters; and the set intersection and set subtraction operations, which restrict the candidate characters to their intersection with a given set of symbols (`cin`) or to exclude a given set of symbols (`cmi`).

The choice of a strict greater than ($>$) and non-strict less than ($\leq$) for each category is based purely on a desire for the presented restriction operations to all include a complementary operation (i.e., equality and disequality, and set intersection and set subtraction). The pairs could just as well have been $<$ and $\geq$, $<$ and $>$, or $\leq$ and $\geq$.

10.2.2 Restriction Operations on Bounded-length Strings

Naturally, some of the restriction operations in Table 10.1 may be unsuited to a particular choice of representation for string variable domains.

In particular, consider a string variable domain represented by a b -length sequence as defined in Chapter 8. A b -length sequence is an over-approximation, i.e., a closure operation that maps one set of strings to a potentially larger set. Removing an arbitrary string from this closed over-approximation will sometimes result in no change to the b -length sequence, and hence no change to the concrete set of strings it represents. A restriction operation implicitly guarantees that the requested restriction will be executed, and that the intended values will no longer be present in the affected variable domain; without this guarantee, the correctness of propagator algorithms cannot be ensured.

Hence, a variable implementation with a b -length sequence as a domain representation cannot include the string disequality and intersection restriction operations. If the set of candidate lengths is over-approximated by an integer interval, as described in Chapter 8, then the length disequality and intersection operations are similarly impermissible. Sets of candidate characters are assumed to be explicitly represented, in which case the character disequality and intersection operations are allowable.

Several of the remaining restriction operations can be eliminated by rewriting them using other operations. For example, the `sle` operation has a decomposition into a conjunction of `cle` and `lle` operations, and the `sgr` operation into a conjunction of `cgr` and `lgr` operations. All of the set intersection and set subtraction operations are equivalent to a series of disequality operations from the same category (e.g., an arbitrary set intersection or subtraction is equivalent to a conjunction of disequalities over the removed values).

Of course, it may be more efficient to implement some of these decomposable restriction operations directly.³ For the purpose of *defining* the restriction operations for domains represented by b -length sequences, though, a set of four required operations is sufficient: `lle`, `lgr`, `ceq`, and `cnq`. These four basic restriction operations, which are required for any variable implementation with a bounded-length sequence domain representation, allow a propagator to increase the lower bound of the string length (`lgr`), decrease the upper bound of the string length (`lle`), fix the character at an index i (`ceq`), or forbid a character at index i (`cnq`).⁴

10.2.3 Equality Semantics

The character restriction operations of Table 10.1 implicitly depend on an indexing operation that maps a string-index pair to a symbol. For finite strings, this indexing operation is a partial function:

$$x[i] = \begin{cases} i\text{-th character of } x & \text{if } 1 \leq i \leq |x| \\ \perp & \text{otherwise} \end{cases}$$

where $\perp$ means that the result is undefined. In order to make the character restriction operations semantically unambiguous, we now explicitly describe the handling of this undefinedness.⁵

We consider three possible semantics of the `ceq` and `cnq` restriction operations. All three are summarized in Table 10.2, in which the restriction operations are defined in terms of a logical proposition over an element x of the domain $\text{dom}'(X_j)$ of a string variable X_j *after* the restriction operation is applied to $\text{dom}(X_j)$.

The obvious choice is whether restricting a set of candidate characters has an effect on the candidate lengths and vice versa. Under the *weak equality* semantics, candidate characters and candidate lengths are connected only insofar as an empty set of candidate characters represents a forbidden index; however, neither a `ceq` nor a `cnq` restriction operation on character i implies that i is a lower bound of the resulting set of candidate lengths. Under the *strict equality* semantics, both the `ceq` and the `cnq` restriction operations on character i also restrict the candidate lengths, removing all strings of length less than i .

³Replacing a single restriction operation with several equivalent ones is, in fact, even worse than it seems, as it additionally burdens the CP solver's propagation services.

⁴In practice, `cin` is a better choice than `cnq`, as it is easier to efficiently implement the latter with the former than the other way around: `cnq` was chosen here as it better illustrates the relationship between equality and disequality in Section 10.2.3.

⁵The semantics of undefinedness in regards to restriction operations is closely related to the semantics of undefinedness in constraint models, as addressed in [35]; indeed, the discussion of equality semantics in this section is directly inspired by [35]. However, the concerns at these two levels differ substantially, a topic that will be developed more thoroughly later in this section.

Table 10.2. *Three semantic interpretations of equality and disequality restriction operations for string variable X_j .*

semantics	$\text{dom}'(X_j) :=$ $\text{ceq}(\text{dom}(X_j), i, c)$ where $\forall x \in \text{dom}'(X_j) :$	$\text{dom}'(X_j) :=$ $\text{cnq}(\text{dom}(X_j), i, c)$ where $\forall x \in \text{dom}'(X_j) :$
weak equality	$ x < i \vee x[i] = c$	$ x < i \vee \neg(x[i] = c)$
strict equality	$ x \geq i \wedge x[i] = c$	$ x \geq i \wedge \neg(x[i] = c)$
strong equality	$ x \geq i \wedge x[i] = c$	$ x < i \vee \neg(x[i] = c)$

Each of these equality semantics seems reasonable; however, as we will shortly demonstrate, neither results in suitable restriction operations. Thus, we introduce a third option: under the *strong equality* semantics, the ceq restriction operation acts like its counterpart from the strict equality semantics, while the cnq restriction operation acts like its counterpart from the weak equality semantics. Its initial oddness aside, the strong equality semantics will be shown to be the most suitable for defining restriction operations.

The evaluation of the propositions in Table 10.2 depends on the interpretation of undefinedness in the string indexing operation. We rely on the three-valued propositional logic of Kleene [55], in which truth values are T, F, and $\perp$. Where our previous use of $\perp$ stood for an undefined symbol, here $\perp$ represents an unknown truth value, which could be either T (for truth) or F (for falsity) in two-valued propositional logic; we tolerate this moderate overloading of the symbol, as there is little danger of confusion. In Kleene’s three-valued logic, $T \vee \perp$ is T because it would be true if $\perp$ stood for either T or F; but $T \wedge \perp$ is $\perp$ because it would be T if $\perp$ were T, and F if $\perp$ were F. The truth tables for the logical connectives $\wedge$, $\vee$, and $\neg$ of this logic are:

$\wedge$	T	F	$\perp$	$\vee$	T	F	$\perp$	$\neg$	
T	T	F	$\perp$	T	T	T	T	T	F
F	F	F	F	F	T	F	$\perp$	F	T
$\perp$	$\perp$	F	$\perp$	$\perp$	T	$\perp$	$\perp$	$\perp$	$\perp$

We evaluate a statement of the form $x[i] = c$ as $\perp$ if and only if $x[i]$ is $\perp$;⁶ otherwise, $x[i] = c$ is T if $x[i]$ and c are the same symbol, and F if they are not. Under this truth-conditional semantics, the propositions from Table 10.2 take the truth values shown in Table 10.3. Due to the relationship between string length, character index, and undefinedness, there are only three possible combinations of truth values for the propositions $|x| \geq i$ and $x[i] = c$: if the length of x is at least i , then $x[i]$ is defined and $x[i] = c$ must be either T or F, otherwise it is $\perp$.

⁶As restriction operations are used exclusively by propagator algorithms, which supply (defined) values upon their invocation, we have no need to allow c to be $\perp$.

Table 10.3. *Truth table for string variable restriction operation semantics*

α $ x \geq i$	β $x[i] = c$	weak equality		strong equality		strict equality	
		ceq $\neg\alpha \vee \beta$	cnq $\neg\alpha \vee \neg\beta$	ceq $\alpha \wedge \beta$	cnq $\neg\alpha \vee \neg\beta$	ceq $\alpha \wedge \beta$	cnq $\alpha \wedge \neg b$
T	T	T	F	T	F	T	F
T	F	F	T	F	T	F	T
F	$\perp$	T	T	F	T	F	F

Looking at Table 10.3, it is worth noting that, under all three equality semantics, both restriction operations are well-defined (i.e., either T or F) for all string and index combinations, including those for which the indexing operation results in $\perp$. Also, we note that when the result of the string indexing operation is defined for the index i the three equality semantics agree. In their handling of undefined characters, though, all three semantics are different.

Restriction Operations as Update Rules

Table 10.4 defines, for string variable domains represented by a b -length sequence, the four chosen restriction operations. The restriction operations are given as a series of update rules on the components of the affected b -length sequence; for each restriction operation, only those components of the resulting b -length sequence $\langle \vec{\mathcal{A}}^b, \mathcal{N}' \rangle$ that differ from the corresponding component of the initial b -length sequence are defined; all other components are unchanged.

All of the restriction operations in Table 10.4 are designed to respect the representation invariant of the b -length sequence representation, which enforces the relationship between empty sets of candidate characters and the upper bound of the set of candidate lengths. Hence, the restriction operation $\text{ll}_{\text{e}}^*(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$, which reduces the upper bound of $\mathcal{N}$, also updates all sets of candidate characters at indices greater than ℓ to be the empty set. The restriction operation $\text{lg}_{\text{r}}^*(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$ requires no corresponding update on sets of candidate characters with indices less than or equal to ℓ : if there existed an empty set $\mathcal{A}[i] = \emptyset$ of candidate characters such that i was less than or equal to ℓ , then the upper bound of $\mathcal{N}$ would already be at most i ; hence $\mathcal{N} \cap [\ell + 1, b]$ would also be the empty set, and the domain would be failed.

The two character restriction operations are defined using each of the proposed equality semantics. In the remainder of this section, each of the three equality semantics is explored in turn, in order to determine which is most suitable for implementing the restriction operations.

Weak Equality Semantics

Under the weak equality semantics, the only connection between the set of candidate lengths and the sets of candidate characters is that imposed by the representation invariant: the index of any empty set of candidate characters

Table 10.4. Comparison of restriction operations implementing the weak equality semantics (w), strict equality semantics (d), and strong equality semantics (s), for string variables implemented by a b-length sequence representation. The two length restriction operations are the same in all three equality semantics; hence we denote them as lle^* and lgr^* . Primed set identifiers ($\mathcal{A}'[i]$ and $\mathcal{N}'$) indicate changed component values in the b-length sequence resulting from the restriction operation; all other components in the resulting b-length sequence are identical to the corresponding component in the original b-length sequence.

restriction operation	update rule
$\text{lle}^*(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$	forall $k \in [\ell + 1, b] : \mathcal{A}'[k] := \emptyset;$ $\mathcal{N}' := \mathcal{N} \cap [0, \ell]$
$\text{lgr}^*(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, \ell)$	$\mathcal{N}' := \mathcal{N} \cap [\ell + 1, b]$
$\text{ceq}^w(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] := \mathcal{A}[i] \cap \{c\};$ if $\mathcal{A}'[i] = \emptyset$ then $(\mathcal{N}' := \mathcal{N} \cap [0, i - 1];$ forall $k \in [i, b] : \mathcal{A}'[k] := \emptyset)$
$\text{cnq}^w(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] := \mathcal{A}[i] \setminus \{c\};$ if $\mathcal{A}'[i] = \emptyset$ then $(\mathcal{N}' := \mathcal{N} \cap [0, i - 1];$ forall $k \in [i, b] : \mathcal{A}'[k] := \emptyset)$
$\text{ceq}^d(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] := \mathcal{A}[i] \cap \{c\};$ $\mathcal{N}' := \mathcal{N} \cap [i, b]$
$\text{cnq}^d(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] := \mathcal{A}[i] \setminus \{c\};$ $\mathcal{N}' := \mathcal{N} \cap [i, b]$
$\text{ceq}^s(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] := \mathcal{A}[i] \cap \{c\};$ $\mathcal{N}' := \mathcal{N} \cap [i, b]$
$\text{cnq}^s(\langle \vec{\mathcal{A}}^b, \mathcal{N} \rangle, i, c)$	$\mathcal{A}'[i] := \mathcal{A}[i] \setminus \{c\};$ if $\mathcal{A}'[i] = \emptyset$ then $(\mathcal{N}' := \mathcal{N} \cap [0, i - 1];$ forall $k \in [i, b] : \mathcal{A}'[k] := \emptyset)$

must be greater than the maximum candidate length. Definitions for the two character restriction operations under the weak equality semantics are given in Table 10.4 as ceq^w and cnq^w .

The weakness of the ceq^w restriction operation is shown in the following example:

Example 10.1

Suppose that the domain of the string variable X_j is represented by the b -length sequence $\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle$ where $b = 4$, as follows:

$$\langle \mathcal{A}_j[1], \dots, \mathcal{A}_j[b], \mathcal{N}_j \rangle = \langle \langle \{1, 2\}, \{1, 2\}, \{5, 6\}, \emptyset \rangle, [1, 3] \rangle$$

Consider the following restriction operation, which removes from the domain of X_j any string x that fails to satisfy $x[3] = 2$:

$$\langle \vec{\mathcal{A}}_j'^4, \mathcal{N}_j' \rangle := \text{ceq}^w(\langle \vec{\mathcal{A}}_j^4, \mathcal{N}_j \rangle, 3, 2)$$

As can be seen in Table 10.4, the third set of candidate characters is updated:

$$\mathcal{A}_j'[3] := \mathcal{A}_j[3] \cap \{2\} = \{5, 6\} \cap \{2\} = \emptyset$$

Since $\mathcal{A}_j'[3]$ is the empty set, the following updates also apply:

$$\begin{aligned} \mathcal{N}_j' &:= \mathcal{N}_j \cap [0, 3 - 1] = [1, 2] \\ \text{forall } k \in [3, b] : \mathcal{A}_j'[k] &:= \emptyset \end{aligned}$$

The domain represented by the resulting b -length sequence is computed applying (8.2) and (8.13):

$$\begin{aligned} \text{dom}'(X_j) &= \gamma^{\mathbb{S}_b}(\vec{\mathcal{A}}_j'^4, \mathcal{N}_j') = \bigcup_{\ell \in [1, 2]} \left\{ w \in \Sigma^\ell \mid \forall k \in [1, \ell] : w[k] \in \mathcal{A}_j'[k] \right\} \\ &= \{ '1', '2', '11', '12', '21', '22' \} \end{aligned}$$

Intuitively, the restriction operation $\text{ceq}^w(\langle \vec{\mathcal{A}}_j^4, \mathcal{N}_j \rangle, 3, 2)$ should result in a domain where every string has the symbol '2' as a third character, a property exhibited by none of the strings in $\text{dom}'(X_j)$.

Allowing restriction operations defined under the weak equality semantics to be used in propagator algorithms also makes the propagator algorithms partly responsible for the connection between sets of candidate lengths and sets of candidate characters, a design choice that results in a poor separation of concerns.

The weak equality semantics is a suitable interpretation of restriction operations when considering how the variable implementation itself is allowed to restrict the domain representation. Any restriction to the variable domain data structure that *does* respect the representation invariant also has a decomposition into several weak-equality-semantics restriction operations. The variable implementation can implement any restriction operation as such a decomposition, as long as the final result still maintains the representation invariant.

Strict Equality Semantics

Under the strict equality semantics, both the disequality and equality restriction operations also make the affected character a mandatory character. Definitions for the two character restriction operations under the strict equality semantics are given in Table 10.4, where they are denoted ceq^d and cnq^d .

The unsuitability of the strict equality semantics restriction operations is immediately apparent: these character restriction operations provide no way for a propagator algorithm to restrict the set of candidate characters for an optional character, without first making that character mandatory. In other words, under the strict equality semantics, removing a symbol from the set of candidate characters in the optional region has the effect of removing *all* strings from that domain that are not long enough to include a character at that index. This effect would severely restrict the inference that a propagator algorithm is able to perform: the number of optional characters can be decreased, and restrictions can be placed on sets of candidate characters in the mandatory region, but no other pruning is possible.

While the strict equality semantics is clearly unsuitable for defining restriction operations, it is a reasonable choice when defining constraints. CP models containing expressions that could result in undefined values, called *unsafe* models, are handled differently by different CP solvers, with results that are frequently at odds with the intuition of modelers [35]. The string indexing operation can be viewed as a special case of an array lookup operation, which as a partial function is a frequent source of undefinedness in CP models. Three different semantic accounts of undefinedness in constraint models are described in [35], each with a method for transforming unsafe models into safe models that avoid undefined values. If applied to constraints rather than restriction operations, the strict equality semantics described here corresponds to the strict three-valued logic semantics of that paper; similarly, the strong equality semantics would, at the modeling level, correspond to the Kleene semantics, while the weak equality semantics would be an example of a relational semantics. The transformation methods described in [35], while not necessary in the context of restriction operations,⁷ can be used at the modeling level to transform an unsafe string constraint model into a safe, yet equivalent, string constraint model. This transformation, while possible under any of the three

⁷As noted previously, none of the restriction operations have undefined results (see Table 10.3).

semantics, is easiest for the strict semantics, which is therefore an appealing choice of semantics at the modeling level.

Strong Equality Semantics

Under the strong interpretation, the character equality restriction operation also makes the affected character a mandatory character, just like under the strict semantics, while like the weak equality semantics the character disequality restriction operation does not force the affected character to be a mandatory character. Definitions for the two character restriction operations under the strong equality semantics are given in Table 10.4, where they are denoted as ceq^s and cnq^s . The restriction operations defined under this equality semantics completely maintain the structural link between the set of candidate lengths and the sets of candidate characters.

Example 10.2

Consider the following strong-equality semantics restriction operation on the initial b -length sequence $\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle$ from Example 10.1, where $b = 4$:

$$\langle \vec{\mathcal{A}}_j^{\prime\prime 4}, \mathcal{N}_j^{\prime\prime} \rangle := \text{ceq}^s \left(\langle \vec{\mathcal{A}}_j^4, \mathcal{N}_j \rangle, 3, 2 \right)$$

This restriction operation is implemented by the following updates:

$$\begin{aligned} \mathcal{A}_j^{\prime\prime}[3] &:= \mathcal{A}_j[3] \cap \{2\} = \{5, 6\} \cap \{2\} = \emptyset \\ \mathcal{N}_j^{\prime\prime} &:= \mathcal{N}_j \cap [3, 4] = [1, 3] \cap [3, 4] = [3, 3] \end{aligned}$$

The result of this restriction operation differs from that of Example 10.1:

$$\text{dom}^{\prime\prime}(X_j) = \gamma^{\mathbb{S}_b} \left(\vec{\mathcal{A}}_j^{\prime\prime 4}, \mathcal{N}_j^{\prime\prime} \right) = \bigcup_{\ell \in \{3\}} \left\{ w \in \Sigma^\ell \mid \forall k \in [1, \ell] : w[k] \in \mathcal{A}_j^{\prime\prime}[k] \right\}$$

The resulting domain contains no strings, as 3 is the only candidate length, yet there are no candidate characters at index 3.

At first glance, the equality and disequality restriction operations seem an odd pairing under the strong equality semantics: the b -length sequences resulting from $\text{ceq}^s(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, i, c)$ and $\text{cnq}^s(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, i, c)$ partition the set $\mathcal{A}_j[i]$ of candidate characters into the sets $\{c\}$ and $\mathcal{A}_j[i] \setminus \{c\}$, but they fail to similarly partition the set of candidate lengths. However, these two restriction operations do define a partition of the corresponding string variable domain:

$$\begin{aligned} &\gamma^{\mathbb{S}_b} \left(\text{ceq}^s \left(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, i, c \right) \right) \cup \gamma^{\mathbb{S}_b} \left(\text{cnq}^s \left(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, i, c \right) \right) = \gamma^{\mathbb{S}_b} \left(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle \right) \\ \text{and} \quad &\gamma^{\mathbb{S}_b} \left(\text{ceq}^s \left(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, i, c \right) \right) \cap \gamma^{\mathbb{S}_b} \left(\text{cnq}^s \left(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, i, c \right) \right) = \emptyset \end{aligned}$$

Example 10.3

Consider the following pair of restriction operations on the initial b -length sequence $\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle$ from Examples 10.1 and 10.2, where $b = 4$:

$$\begin{aligned}\langle \vec{\mathcal{A}}_j'^4, \mathcal{N}_j' \rangle &:= \text{ceq}^s \left(\langle \vec{\mathcal{A}}_j^4, \mathcal{N}_j \rangle, 3, 6 \right) \\ \langle \vec{\mathcal{A}}_j''^4, \mathcal{N}_j'' \rangle &:= \text{cnq}^s \left(\langle \vec{\mathcal{A}}_j^4, \mathcal{N}_j \rangle, 3, 6 \right)\end{aligned}$$

The resulting sets $\mathcal{A}_j'[3]$ and $\mathcal{A}_j''[3]$ of candidate characters form a partition of $\mathcal{A}_j[3]$:

$$\begin{aligned}\mathcal{A}_j'[3] &:= \mathcal{A}_j[3] \cap \{6\} = \{6\} \\ \mathcal{A}_j''[3] &:= \mathcal{A}_j[3] \setminus \{6\} = \{5\}\end{aligned}$$

The resulting sets $\mathcal{N}_j'$ and $\mathcal{N}_j''$ do not, however, form a partition of $\mathcal{N}_j$:

$$\begin{aligned}\mathcal{N}_j' &:= \mathcal{N}_j \cap [3, 4] = [3, 3] \\ \mathcal{N}_j'' &:= \begin{cases} \mathcal{N}_j \cap [0, 3-1] & \text{if } \mathcal{A}_j'[3] = \emptyset \\ \text{otherwise} \end{cases} = [1, 3]\end{aligned}$$

However, the domains represented by these two b -length sequences do in fact form a partition of the original domain:

$$\begin{aligned}\text{dom}(X_j) &= \gamma^{\mathbb{S}_b} \left(\langle \vec{\mathcal{A}}_j^4, \mathcal{N}_j \rangle \right) \\ &= \bigcup_{\ell \in [1, 3]} \left\{ w \in \Sigma^\ell \mid \forall k \in [1, \ell] : w[k] \in \mathcal{A}_j[k] \right\} \\ &= \left\{ \begin{array}{l} '1', '2', '11', '12', '21', '22', \\ '115', '116', '125', '126', \\ '215', '216', '225', '226' \end{array} \right\} \\ \text{dom}'(X_j) &= \gamma^{\mathbb{S}_b} \left(\langle \vec{\mathcal{A}}_j'^4, \mathcal{N}_j' \rangle \right) \\ &= \bigcup_{\ell \in \{3\}} \left\{ w \in \Sigma^\ell \mid \forall k \in [1, \ell] : w[k] \in \mathcal{A}_j'[k] \right\} \\ &= \{ '116', '126', '216', '226' \} \\ \text{dom}''(X_j) &= \gamma^{\mathbb{S}_b} \left(\langle \vec{\mathcal{A}}_j''^4, \mathcal{N}_j'' \rangle \right) \\ &= \bigcup_{\ell \in [1, 3]} \left\{ w \in \Sigma^\ell \mid \forall k \in [1, \ell] : w[k] \in \mathcal{A}_j''[k] \right\} \\ &= \left\{ \begin{array}{l} '1', '2', '11', '12', '21', '22', \\ '115', '125', '215', '225' \end{array} \right\}\end{aligned}$$

Equality Semantics for Bounded-Length String Restriction Operations

In light of the preceding analysis, it appears that the strong equality semantics yields the most suitable interpretation of string equality and disequality in the context of string variable restriction operations. The character equality and disequality restriction operations in the strong equality semantics are perhaps the least intuitive of the three choices when considered from the perspective of the b -length sequences that they act upon, yet the two operations are in fact complementary in so far as they result in a partition of the represented string variable domain. This property, while not required of restriction operations, is both appealing and useful. Consider, for example, branching choices: as described in Chapter 1, a choice is a constraint, the implementation of which will rely on restriction operations, and the set of choices at any node must form a complete partition of the domain.

The differences in the three semantics considered here can be summed up as viewing string equality from three different perspectives. A variable implementation acts directly on the individual components of the data structure representing its domain; these operations express the semantics of weak equality, in which candidate lengths and candidate characters are distinct. A propagator algorithm prunes candidate values (in this case, strings) from the domains of variables; these operations express the semantics of strong equality, respecting the relationship between candidate lengths and candidate characters that is formalized by the representation invariant. A CP model expresses a CSP as variables, domains, and constraints; under the strict equality semantics, unsafe models are more easily transformed into safe ones.

In the implementation that we evaluate in Chapter 11, our string variable supplied the restriction operations `ceqs`, `cnqs`, `lle*`, `lgr*`, and `cins`. Several other restriction operations were supplied for convenience, but implemented using the five listed operations.

10.3 Data Structure

Having decided on a set of restriction operations for the string variable implementation, we now know what operations need to be efficiently supported by the data structure for the corresponding domain. Thus, we proceed by considering how this data structure might be implemented.

For a bounded-length string variable type with a domain represented by a b -length sequence, there are three largely orthogonal structural choices to be considered: the representation of the set $\mathcal{N}$ of candidate lengths, the representation of the sets $\mathcal{A}[i]$ of candidate characters, and the construction of the sequence $\langle \mathcal{A}[1], \dots, \mathcal{A}[b] \rangle$ itself. Each of these choices is now considered in turn. Before proceeding, however, we first describe different choices of representations for sets of natural numbers, as these choices will form a common

Table 10.5. *Worst-case complexity for fundamental operations on sets $\mathcal{S}_1, \mathcal{S}_2 \subset \mathbb{N}$ for implementations of $\mathcal{S}_1, \mathcal{S}_2$ as range sequences of length r_1, r_2 , bitsets of k words of w bits, or interval over-approximations.*

operation	range sequence	bitset	interval
$v \in \mathcal{S}_1$	$\mathcal{O}(r_1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\min(\mathcal{S}_1)$	$\mathcal{O}(1)$	$\mathcal{O}(kw)^b$	$\mathcal{O}(1)$
$\max(\mathcal{S}_1)$	$\mathcal{O}(1)$	$\mathcal{O}(kw)^b$	$\mathcal{O}(1)$
$ \mathcal{S}_1 $	$\Theta(r_1)$	$\mathcal{O}(kw)^b$	$\mathcal{O}(1)$
$\mathcal{S}_1 \neq \emptyset$	$\mathcal{O}(1)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$
$\mathcal{S}_1 \setminus \{v\}$	$\mathcal{O}(r_1)$	$\mathcal{O}(k)$	$\mathcal{O}(1)^a$
$\{i \in \mathcal{S}_1 \mid i \leq v\}$	$\mathcal{O}(r_1)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$
$\{i \in \mathcal{S}_1 \mid i > v\}$	$\mathcal{O}(r_1)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$
$\mathcal{S}_1 \cap \mathcal{S}_2$	$\mathcal{O}(r_1 + r_2)$	$\mathcal{O}(k)$	$\mathcal{O}(1)$
$\mathcal{S}_1 \cup \mathcal{S}_2$	$\mathcal{O}(r_1 + r_2)$	$\mathcal{O}(k)$	$\mathcal{O}(1)^a$
$\mathcal{S}_1 + \mathcal{S}_2^c$	$\mathcal{O}(r_1 \cdot r_2)$	$\mathcal{O}(k^2w)$	$\mathcal{O}(1)$

^a operation over-approximates results

^b more efficient, hardware-dependent operations exist

^c Minkowski sum

background to the consideration of both sets of candidate lengths and sets of candidate characters, to follow.

Representations for Sets of Natural Numbers

Two common choices for implementing the domains of integer variables are range sequences and bitsets; another, less common choice is to over approximate the set of feasible values by an interval, as described in Section 5.1.

A *range sequence* $\mathcal{R}$ for a finite set $\mathcal{S} \subset \mathbb{N}$ is the smallest sequence of integer intervals

$$\mathcal{R} = \langle \langle \ell_1, u_1 \rangle, \dots, \langle \ell_r, u_r \rangle \rangle$$

such that the intervals are ordered by lower bound (i.e., $\ell_i < \ell_{i+1}$ for all $i \in [1, r-1]$), are non-overlapping ($u_i < \ell_{i+1}$ for all $i \in [1, r-1]$) and non-adjacent ($u_i + 1 \neq \ell_{i+1}$ for all $i \in [1, r-1]$), and $\mathcal{S}$ is covered ($\mathcal{S} = \bigcup_{i=1}^r [\ell_i, u_i]$). Table 10.5 lists the computational complexity of several basic operations, relevant for domain representations, on a range sequence implementation of a natural-number set. The range sequence is assumed to be implemented as a doubly-linked list. Set operations (i.e., intersection, union, etc.) are both easily implemented and efficient on range sequences, with the exception of the Minkowski sum operator.

A *bitset* for a finite set $\mathcal{S} \subset \mathbb{N}$ is a sequence of $\max(\mathcal{S})$ bits (i.e., values in $\{0, 1\}$) such that the i -th bit is equal to 1 if and only if $i \in \mathcal{S}$. On a machine with word size w , a bitset representation of $\mathcal{S}$ requires $k = \lceil \max(\mathcal{S})/w \rceil$ words, or

a total of kw bits. As shown in Table 10.5, bitsets allow for very fast set operations (typically, a small constant number of operations per word is required), but only if all involved bitsets use the same k and w for their representation. Note that in practice, depending on hardware, the complexity of some listed operations can be substantially reduced by taking advantage of more efficient built-in operations.

Table 10.5 also includes interval over-approximations. Intervals are extremely time and space efficient; however, the over-approximation is not tight, especially for sparse sets. It is worth noting, though, that intervals are the only considered representation that efficiently handles the Minkowski sum operation.

Range sequences generally scale to large domain sizes better than bitsets [86]; for domains taken from small sets of candidate solutions, though, bitsets are extremely space efficient.

10.3.1 Length

The choice of representation for a set $\mathcal{N}$ of candidate lengths, for a b -length sequence representing the domain of a string variable X , depends on two factors: first, how efficient it is to implement the types of length operations required by propagation algorithms implementing string constraints, and second, how important it is to have an exact representation of the set of candidate lengths.

The length restriction operations `lles` and `lgrs` have the effect of tightening the upper bound and lower bound, respectively, of a set of candidate lengths. These operations may be efficiently implemented on any of the considered representations, as shown in Table 10.5. As shown in Section 10.1.1, however, a propagator algorithm implementing the constraint `CAT` calculates the Minkowski sum of the sets of candidate lengths of the string variables in its scope. Computing the Minkowski sum of two sets is essentially quadratic in the length of the set representation, that is, in the number of elements, intervals, or bounds used to represent the set. As Table 10.5 shows, computation of the Minkowski sum is trivial on the interval representation, as it collapses to a pair of sums on the upper and lower bounds of the operands; however, using either the range sequence or bitset implementation it has a substantially higher worst-case complexity.

The insight of Section 10.1.3 suggests that the upper and lower bounds of the set of candidate lengths are more useful than interior values, which also argues in favor of an interval implementation. Most of the string constraints defined in Section 7.2 afford only limited opportunity for interior reasoning on length intervals. For example, none of the binary pure string constraints from Section 7.2.1 benefits from interior reasoning on sets of candidate lengths.

Given, on the one hand, the importance of the Minkowski sum operation to the length arithmetic portion of the propagate algorithm for the `CAT` constraint,

and, on the other hand, the apparent dearth of string constraints with efficient propagation algorithms that perform non-trivial reasoning on the interior values of a set $\mathcal{N}$ of candidate lengths, it seems that an integer interval representation is the most suitable choice of representation for $\mathcal{N}$. Thus, that is the representation we chose for the implementation that is evaluated in Chapter 11.

10.3.2 Symbols

While an interval representation is reasonable for the set of candidate lengths, mapping $\mathcal{A}[i]$ to an interval on $\mathbb{N}$ is not ideal. This representation would allow for inexpensive intersection operations, but, as discussed in Section 5.1, the least upper bound of two intervals is the convex hull of the union of the corresponding sets, resulting in significantly weaker propagation. A range sequence, on the other hand, allows for an exact representation, at the cost of increased memory usage for sparse sets, and increased complexity for both union and intersection. More importantly, the insight that intervals in the alphabet Σ lack meaning (see Section 10.1.3) argues against both intervals and range sequences. When a propagator for a string constraint removes a set $\mathcal{S} \subseteq \Sigma$ of symbol values from a set $\mathcal{A}[i]$ of candidate characters, the set $\mathcal{S}$ may form an interval, but only incidentally. Removing an arbitrary set $\mathcal{S}$ from $\mathcal{A}[i]$ will result in either minimal pruning on an interval representation, or the creation of a relatively large number of holes on a range sequence representation.

As the cardinality of Σ is generally manageable, a bitset implementation seems reasonable, and is therefore the representation we have chosen. On a machine with word size w , a bitset representation of Σ requires $k = \lceil \max(\Sigma)/w \rceil$ words, or a total of kw bits. The time complexity of set operations such as union and intersection, as shown in Table 10.5, is linear in k , while membership queries take constant time. Furthermore, as long as $|\Sigma|$ is not significantly larger than w , the memory requirement of a bitset representation of a dense set is not worse than that for a range sequence representation; for a set with many holes, a bitset will have substantially better memory usage.

On the other hand, operations such as determining the cardinality of a set, or enumerating the symbols contained therein, are linear in $|\Sigma|$ in the worst case. In practice, this is not a significant limitation: on many platforms the variable implementation will be able to take advantage of more efficient hardware implementations of these operations, made available at compile time via *built-in*, or *intrinsic*, functions (e.g., [93]).

10.3.3 Sequence

The data structure for representing the sequence $\langle \mathcal{A}[1], \dots, \mathcal{A}[b] \rangle$ of sets of candidate characters remains to be decided. There are two choices to be made: how to structure the sequence, and how to handle the allocation of its contents.

Sequence Structure

The two basic options are an array-based structure or a list-based structure. Each has well-known advantages and disadvantages: an array has minimal memory overhead and affords efficient access to the elements, while a list allows for better memory management when the size of the sequence changes frequently.

One key factor in the selection of a sequence structure is the allocation strategy, discussed below.

Another key factor is the amount of memory overhead: for small alphabet sizes and sets of candidate characters implemented as bitsets, a large portion of a linked list would be occupied by pointers. Since the size of the data stored in each node of the list is expected to be small, an *unrolled* linked list seems like a reasonable choice. Each node of an unrolled linked list holds a *block*, that is, a fixed-length array of bitsets.

One potential enhancement is suggested by observing that the maximum length of any b -length sequence is already known, namely b . Thus, rather than a list of blocks, the sequence of sets of candidate characters could be implemented as a fixed-length array of n pointers to blocks of m bitsets each, where $n = \lceil \frac{b}{m} \rceil$. This design overcomes one drawback of linked lists, namely that accessing nodes in the middle of the list requires traversing the list from the beginning (or alternately the end, in the case of a doubly-linked list). The tradeoff is a slightly larger memory overhead for short sequences when compared with the unrolled list, as the former maintains a fixed number of pointers and the latter does not.

Allocation Strategy

There are many strategies for deciding when to allocate memory for storing sets of candidate characters. We consider three in particular:

- The *eager* strategy is to initially allocate enough memory to accommodate as many sets of candidate characters as would be required to represent a string of the maximum allowed size; for a b -length sequence, this means allocating memory for b sets of candidate characters. Upon execution of a restriction operation, no additional allocation is required.
- The *extremely lazy* strategy is to initially allocate no memory. Upon execution of a restriction operation at index i , memory sufficient to accommodate all sets of candidate characters in the interval $[1, i]$ is allocated, extending the list to (at least) the affected index.
- The *lagom*⁸ *lazy* strategy differentiates between mandatory sets of candidate characters and optional sets of candidate characters. Some symbol from every mandatory set of candidate characters must participate in any string that is a candidate solution. So it is reasonable to always allocate

⁸The Swedish word *lagom* means “just the right amount,” and is quintessential to Swedish culture.

enough storage for these sets of candidate characters, without waiting for a restriction operation on these indices. Thus, the initial allocation under the *lagom* lazy strategy is to allocate memory sufficient to accommodate $\min(\mathcal{N})$ sets of candidate characters. Upon a restriction operation at an optional index, or after an increase to the lower bound of the set $\mathcal{N}$ of candidate lengths, additional memory may be allocated, as per the extremely lazy strategy.

Deallocation of sets of candidate characters could be performed whenever the upper bound of the set $\mathcal{N}$ of candidate lengths decreases, thus freeing memory used by sets of candidate characters that will never participate in any satisfying string. As discussed in Section 10.1.2, though, GECODE is a copying-based CP solver, rather than a trailing-based CP solver. This implies a natural alternative deallocation strategy: whenever a copy is made of a node, the memory allocated for the sets of candidate characters in the copy only needs to be sufficient to accommodate the upper bound of $\mathcal{N}$ at the time of the copy. This rather simple deallocation strategy makes even the eager allocation strategy potentially reasonable.

There are clear arguments to be made for the choice of both an integer interval to represent the set of candidate lengths, and bitsets to represent the sets of candidate characters. The most suitable choice of a structure to represent the sequence, however, is much less clear. In the next chapter, we report the results of an experimental evaluation of four possible sequence representations: a statically allocated array; a dynamically allocated array; a dynamically allocated, singly-linked block list; and a dynamically allocated block array.

10.4 Propagation Events

As discussed in Section 10.1, most CP solvers rely on *propagation events* to communicate to a propagator function how the domain of a particular variable in its scope has changed. In such a solver, the design of a variable type specifies a set of propagation events, called an *event system*. Propagation events can, in principle, communicate more or less detailed information about domain changes; however, defining a larger event system generally comes with a commensurate cost to efficiency (see [88] for a detailed analysis of event-based propagation). So it makes sense to restrict the chosen event system to include only those propagation events that are actually useful: a propagation event is useful to a propagator algorithm if it distinguishes between a change that leaves the propagator algorithm at fixpoint, versus one that does not.

A minimal event system for a string variable X_j with a domain represented by a b -length sequence $\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle$ is shown in Figure 10.3. The `val` propagation event indicates that the string variable is fixed, that is, that there is only one string in the domain. Understandably, given the nature of the chosen domain representation, each of the remaining propagation events indicates one

Figure 10.3. Implications in a minimal string variable event system (transitive implications are omitted for the sake of clarity). If the changes to the set of candidate lengths and the sets of candidate characters are viewed independently (a), then the resulting event system is not monotonic. The corrected event system (b) is monotonic.

of two kinds of change to a domain. The first kind of propagation event indicates a change to the set $\mathcal{N}_j$ of candidate lengths: either the lower bound has increased (LMIN), or the upper bound has decreased (LMAX), or the set is fixed (LVAL). Since the set of candidate lengths is represented by an integer interval, there is no other meaningful information to communicate. The two length-bound events could be combined into a single event representing the fact that at least one bound has changed; however, since each bound has a different relationship to the sets $\vec{\mathcal{A}}_j^b$ of candidate characters, as seen in Section 10.1.3, it seems reasonable to keep them separate. The second kind of propagator event indicates a change to a set $\mathcal{A}_j[i]$ of candidate characters: either some symbol has been removed (CDOM) or the symbol of the character at index i is known (CVAL).

Note that we define CVAL to indicate that the character at some index i is a symbol $c \in \Sigma$, and *not* that the set of candidate characters at i is fixed to $\{c\}$. If i is a mandatory index, then the two conditions are equivalent, and for every string x in the domain of X_j , the character $x[i]$ is equal to c . If i is an optional index, then the fact that the set $\mathcal{A}_j[i]$ of candidate characters is a singleton only indicates that there is just one possible symbol for the character at index i ; for other strings in the domain, the character at index i is undefined.

There is a degree of overlap among the changes described by these propagation events. For example, the change to some domain represented by an LVAL propagation event, namely that the corresponding set of candidate lengths is fixed, implies that at least one of the changes represented by LMIN and LMAX has also occurred. Figure 10.3 shows two interpretations of the implications in the proposed string variable event system. In the first, Figure 10.3(a), the two kinds of propagation events are isolated; intuitively, this makes it possible to report changes to the set of candidate lengths and changes to sets of candidate characters independently. However, this design violates one of the properties required of propagation events [94]: first, a propagation event must describe an actual change to a domain, so no propagation event arises from a restriction

operation that does not change a domain; and second, the change described by a propagator must be monotonic, that is, it must not be undone by later changes.

In a non-monotonic propagation event system, the set of propagation events generated by a series of restriction operations is dependent on their order:

Example 10.4

Let X_j be a string variable with a domain represented by the b -length sequence, where $b = 4$, as follows:

$$\langle \vec{\mathcal{A}}_j^4, \mathcal{N}_j \rangle = \langle \langle \{1, 2\}, \{1, 2\}, \{5, 6\}, \emptyset \rangle, [1, 3] \rangle$$

The following sequence of restriction operations restricts first the set of candidate characters at index 3, and then the set of candidate lengths:

$$\begin{aligned} \langle \vec{\mathcal{A}}_j'^4, \mathcal{N}_j' \rangle &:= \text{lle}^* \left(\text{cnq}^s \left(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, 3, 6 \right), 2 \right) \\ &:= \text{lle}^* \left(\langle \langle \{1, 2\}, \{1, 2\}, \{5, 6\} \setminus \{6\}, \emptyset \rangle, [1, 3] \rangle, 2 \right) \\ &:= \text{lle}^* \left(\langle \langle \{1, 2\}, \{1, 2\}, \{5\}, \emptyset \rangle, [1, 3] \rangle, 2 \right) \\ &:= \langle \langle \{1, 2\}, \{1, 2\}, \{5\} \cap \emptyset, \emptyset \rangle, [1, 3] \cap [0, 2] \rangle \\ &:= \langle \langle \{1, 2\}, \{1, 2\}, \emptyset, \emptyset \rangle, [1, 2] \rangle \end{aligned}$$

The set of propagation events generated by these restriction operations is $\{\text{CDOM}, \text{LMAX}\}$: the set of candidate characters at index 3 was restricted, followed by the set of candidate lengths.

If the order of the two restriction operations is reversed, then the resulting domain representation is the same:

$$\begin{aligned} \langle \vec{\mathcal{A}}_j''^4, \mathcal{N}_j'' \rangle &:= \text{cnq}^s \left(\text{lle}^* \left(\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle, 2 \right), 3, 6 \right) \\ &:= \text{cnq}^s \left(\langle \langle \{1, 2\}, \{1, 2\}, \{5, 6\} \cap \emptyset, \emptyset \rangle, [1, 3] \cap [0, 2] \rangle, 3, 6 \right) \\ &:= \text{cnq}^s \left(\langle \langle \{1, 2\}, \{1, 2\}, \emptyset, \emptyset \rangle, [1, 2] \rangle, 3, 6 \right) \\ &:= \langle \langle \{1, 2\}, \{1, 2\}, \emptyset \setminus \{6\}, \emptyset \rangle, [1, 2] \rangle \\ &:= \langle \langle \{1, 2\}, \{1, 2\}, \emptyset, \emptyset \rangle, [1, 2] \rangle \end{aligned}$$

However, the set of propagation events generated by this sequence of restriction operations is only $\{\text{LMAX}\}$: no CDOM propagation event is generated, because the cnq^s restriction operation did not change the domain.

The monotonicity of event systems is important because propagation events are used in the fixpoint algorithm to schedule propagator algorithms for execution; use of a non-monotonic event system makes propagator scheduling non-monotonic, as well. For a complete explanation of event systems and efficient propagator scheduling, see [89, 94].

Figure 10.3(b) shows a monotonic version of this event system, in which LMAX implies CDOM and CVAL implies LMAX . Intuitively, this implication reflects the effect of the representation invariant: any change to the upper bound of the set $\mathcal{N}_j$ of candidate lengths must empty at least one set $\mathcal{A}_j[i]$ of candidate characters, while an index i changing from optional to fixed (indicated by CVAL) implies that the upper bound of $\mathcal{N}_j$ has decreased, and possibly that some symbol has been removed from $\mathcal{A}_j[i]$ as well.

Other propagation events for string variables could be useful. For example, a propagator algorithm implementing the LEX constraint could take advantage of the information that the upper or lower bound of a set of candidate characters has changed. The set of propagation events presented here, which is sufficient for the implementation of a string variable type based on a b -length sequence domain representation, is reflected in the design of our string variable implementation.

* * *

In the next chapter, we report the results of an experimental evaluation of the string variable type described in this chapter. As previously mentioned, the target solver for that implementation is the GECODE toolkit [96], a mature CP solver with state-of-the-art performance and many deployments in industry. Among GECODE 's finer qualities is its extensive support for the addition of new variable types: custom variable types implemented for GECODE can be directly integrated into the solver kernel, and thus incur no efficiency penalty with respect to built-in types.

11. Experimental Evaluation

In Chapters 9 and 10 we described two very different methods for extending a CP solver to handle bounded-length string variables. These two methods are both based on the b -length sequence representation for sets of strings, defined in Chapter 8. In this chapter, we experimentally evaluate the performance of these two implementations.

The experiments reported in this chapter have two purposes. First, we evaluate some of the choices made in the implementation of the string variable type described in Chapter 10. Second, we compare the performance of that implementation and the aggregate implementation described in Chapter 9, and compare both of those implementations with a control implementation based on the padded-string method, which is the most commonly used method of modeling string constraint problems in CP (e.g., [48]).

The experimental methodology followed in the chapter, including a description of the evaluated implementations and the branching heuristic, is described in Section 11.1. The benchmarks used in the experiments are described in Section 11.2. Finally, the results of the experiments are given in Section 11.3.

11.1 Experimental Methodology

All the experiments reported in this chapter were carried out on a Virtual-Box 4.3.10 virtual client with 1,024 MB of RAM, running Xubuntu 14.04. The host machine was a 2.66 GHz Intel Core 2 Duo with 4 GB of RAM, running OpenSUSE 13.1. Code for implemented propagator algorithms was written in C++ for the GECODE 4.4.0 constraint solving library, and compiled with GCC 4.8.4.

Reported runtimes are the average (including setup time) over 100 runs, using GECODE's benchmark timing mode. Search was sequential, using a deterministic branching heuristic (see Section 11.1.2), with default GECODE recomputation settings.

For each experiment, the same maximum length b was used for every string variable, and all experiments were run for several possible values of b . The timeout for all experiments was set at ten minutes.

11.1.1 Models and Implementations

We compare a total of six implementations. The first four are variants of the bounded-length string variable described in Chapter 10. The fifth is the ag-

gregate implementation described in Chapter 9. The sixth is the *de facto* standard method for solving bounded-length string constraint problems using a CP solver, the padded-string method.

Native-String Method

The native string method makes use of an implementation of the string variable type described in Chapter 10.

Native-String Models

One advantage of the string variable type described in Chapter 10 is that it allows for straightforward modeling of string constraint problems. Every string unknown in a problem can be modeled by a corresponding string variable in GECODE. The string constraints described in Chapter 7 may be used without modification, contingent on the manual development of a corresponding propagator algorithm.

Native-String Implementations

Most of the design decisions explained in Chapter 10 were motivated by concerns arising directly from the nature of strings themselves. In particular, for a string variable X_j with a domain represented by a b -length sequence $\langle \vec{\mathcal{A}}_j^b, \mathcal{N}_j \rangle$, there are clear arguments in favor of the chosen representations for both the set $\mathcal{N}_j$ of candidate lengths (i.e., an integer interval) and the sets $\mathcal{A}_j[i]$ of candidate characters (i.e., bitsets); less clear is the proper choice of data structure for the sequence $\vec{\mathcal{A}}_j^b$ of sets of candidate characters. To determine the effect of the latter choice on overall performance, we evaluate four variants of the string variable implementation:

- In the **static array** implementation, the sequence $\vec{\mathcal{A}}_j^b$ is implemented as a fixed-length array of bitsets. This implementation follows the *static* allocation strategy: initially allocate memory for $\max(\mathcal{N}_j)$ bitsets, on copy allocate memory for $\max(\mathcal{N}_j)$ bitsets, and on a decrease of $\max(\mathcal{N}_j)$ do nothing.
- In the **dynamic array** implementation, the sequence $\vec{\mathcal{A}}_j^b$ is again implemented as a fixed-length array of bitsets. This implementation follows a different allocation strategy, namely a version of the *lagom lazy* allocation strategy: initially allocate memory for $\min(\mathcal{N}_j)$ bitsets, on modification of a set $\vec{\mathcal{A}}_j^b$ of candidate characters allocate additional memory as needed to accommodate a total of j bitsets, on copy allocate memory for the lesser of the current size of the original $\vec{\mathcal{A}}_j^b$ and $\max(\mathcal{N}_j)$, and on a decrease of $\max(\mathcal{N}_j)$ do nothing.
- In the **block list** implementation, the sequence $\vec{\mathcal{A}}_j^b$ is implemented as a series of blocks in a traditional linked-list data structure, composed of nodes containing a single block and a pointer to the next node in the list. This implementation follows a version of the *lagom lazy* allocation

strategy: initially allocate sufficient blocks to accommodate $\min(\mathcal{N}_j)$ bitsets, on modification of a set $\vec{\mathcal{A}}_j^b$ of candidate characters allocate additional blocks as needed to accommodate a total of j bitsets, on copy allocate blocks to accommodate a number of bitsets equal to the lesser of $\max(\mathcal{N}_j)$ and the current size of the original $\vec{\mathcal{A}}_j^b$, and on a decrease of $\max(\mathcal{N}_j)$ deallocate any blocks that are no longer required.

- In the **block array** implementation, the sequence $\vec{\mathcal{A}}_j^b$ is implemented as a fixed-length array of pointers to blocks, where the array of pointers accommodates a sufficient number of blocks to store $\max(\mathcal{N}_j)$ bitsets. This implementation follows the same allocation strategy as the **block list** implementation.

A string variable type serves little purpose without corresponding propagator algorithms. For bounded-length string variables with a b -length sequence domain representation, as described in Chapter 10, the propagator functions derived in Chapter 8 provide a precise description, expressed in terms of a set of candidate lengths and sets of candidate characters, of the pruning that is expected of a propagator algorithm implementing the corresponding constraint. Section 10.1.1 exemplifies this relationship, showing how a derived propagator function relates to restriction operations on string variables. Unfortunately, unlike the aggregate implementation of the b -length sequence representation, for which propagator algorithms can be automatically generated, propagator algorithms for the native-string implementation must be developed manually. Indexical propagator descriptions, such as those obtained in Chapter 9 via automatic synthesis, can serve as a useful starting point in the development of native-string propagator algorithms.

Implementation of the the $\text{REGULAR}_{\mathcal{L}}$ constraint for the native-string implementation requires a different approach. The propagator algorithm used in the following experiments to implement for $\text{REGULAR}_{\mathcal{L}}$ is adapted from the fixed-length GECODE implementation of the algorithm described in [75]. The propagator algorithm has been extended to handle bounded-length sequences using ideas described in [64]. The adapted propagator algorithm makes use of the OPENFST 1.5.0 finite-state transducer library [3] for calculating a table of minimum distances to any accepting state.

Aggregate-String Method

The implementation used in the aggregate-string method is generated using methods described in Chapter 9.

Aggregate-String Models

An aggregate-string implementation, like the native-string implementations described above, can include propagator algorithms for each of the string constraints from Chapter 7, so modeling string constraints is straightforward. Unlike a native-string implementation, there are no string variables in an aggreg-

ate-string implementation. Instead, using an aggregate-string implementation, each string unknown in a problem is modeled as a pessimistically large array of integer variables as described in Section 9.1, allowing multiple occurrences of a null or padding symbol at the end of each string (see Section 9.1.1). An additional scalar integer variable models the set of candidate lengths of the string unknown.

Aggregate-String Implementation

To evaluate the aggregate-string implementation, propagator algorithms implementing the constraints EQ, CAT, SUB, LEN, and REV were automatically generated for GECODE from indexical propagator specifications, following the synthesis-refinement-generation pathway described in Section 9.2, using the tool described in [69].

The exception is the $\text{REGULAR}_{\mathcal{L}}$ constraint, for which no new propagator algorithm is required. Instead, constraints of regular-language membership in the aggregate-string implementation can be handled by transforming the regular language over bounded-length strings into a regular language over fixed-length strings. For a bounded-length string variable X , $\text{REGULAR}_{\mathcal{L}}(X)$ constrains the string variable X to be in the regular language $\mathcal{L}$. Given a null symbol $\lambda \notin \Sigma$, the language of strings from $\mathcal{L}$ followed by zero or more λ symbols, called the λ -padded language of $\mathcal{L}$, is a set of strings over the alphabet $\Sigma \cup \{\lambda\}$, as follows:

$$\mathcal{L}_{\lambda} = \{x \cdot y \mid x \in \mathcal{L}, y \in \lambda^*\} \quad (11.1)$$

Thus, the constraint $\text{REGULAR}_{\mathcal{L}}(X)$ over the string variable X can be rewritten for an aggregate-string model, where X is represented by an array $\vec{X}$ of integer variables, as the constraint $\text{REGULAR}_{\mathcal{L}_{\lambda}}(\vec{X})$.

The λ -padded language of a regular language $\mathcal{L}$ can be automatically generated. For example, let $\mathcal{L}$ be the language over the alphabet Σ that is accepted by the deterministic finite automaton $\mathcal{M}_{\mathcal{L}} = \langle \mathcal{Q}, \Sigma, t, q_0, \mathcal{F} \rangle$ with set of states $\mathcal{Q}$, transition function $t: \mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$, starting state q_0 , and set of accepting states $\mathcal{F} \subseteq \mathcal{Q}$. Then the λ -padded language of $\mathcal{L}$ is accepted by an automaton $\mathcal{M}_{\lambda}$ that is $\mathcal{M}_{\mathcal{L}}$ with one additional accepting state, and a λ -labeled transition from each accepting state of $\mathcal{M}_{\mathcal{L}}$ to the newly added accepting state; that is:

$$\mathcal{M}_{\lambda} = \langle \mathcal{Q} \cup \{q_{\lambda}\}, \Sigma \cup \{\lambda\}, t \cup \{((q, \lambda), q_{\lambda}) \mid q \in \mathcal{F} \cup \{q_{\lambda}\}\}, q_0, \mathcal{F} \cup \{q_{\lambda}\} \rangle$$

If the language $\mathcal{L}$ is instead represented by a regular expression ρ , then $\mathcal{L}_{\lambda}$ is the language accepted by the regular expression $\rho\lambda^*$.

Padded-String Method

The final method considered is the padded-string method, as mentioned in Section 9.1.1, appears to exist more as folk wisdom in the CP community than as a formally recognized method. Nevertheless, as we use the padded-string method to provide control results for the experiments in this chapter, we describe it now in some detail.

```

1 predicate concat(
2   array[int] of var int: X, var int: lenX,
3   array[int] of var int: Y, var int: lenY,
4   array[int] of var int: Z, var int: lenZ
5 ) =
6   lenZ = lenX + lenY /\
7   forall( i in index_set(X) ) (
8     ( i <= lenX ) -> ( X[i] = Z[i] )
9   ) /\
10  forall( i in index_set(X) union {0} ) (
11    forall( j in index_set(Y) ) (
12      ( i = lenX /\ j <= lenY ) -> ( Y[j] = Z[i+j] )
13    )
14  );

```

Figure 11.1. MINIZINC predicate definition for the CAT constraint, by decomposition into primitive integer constraints via the padded-string method.

Padded-String Models

Modeling a string constraint problem using the padded-string method is less straightforward than the preceding two modeling methods. Each string unknown in a problem is modeled by a fixed-length sequence of integer variables (i.e., an integer-variable array), with a length set at a pessimistically large upper bound, and multiple occurrences of a padding symbol are allowed at the end of each string [48], just as described in the aggregate-string method, above.

Unlike the aggregate-string method, in the padded-string method there are no propagator algorithms implementing string constraints. Instead, each string constraint is modeled as a decomposition consisting of a conjunction of reified constraints over the sequence of integer variables corresponding to each string in the scope of the constraint, which express the relationship between the length of the modeled string and the occurrences of the padding character in the corresponding sequence. For example, Figure 11.1 shows a MINIZINC predicate that implements the CAT constraint using the padded-string method. In the MINIZINC modeling language [72], a *predicate* defines a propagator algorithm by decomposition. The `concat` predicate in Figure 11.1 implements the CAT constraint with a `forall` loop over the integer variable array `x` (line 7) and a nested `forall` loop (line 10) over the integer variable arrays `x` and `y`. The resulting decomposition consists of $\frac{n \cdot (n+1)}{2}$ constraints, each of which is conditional upon the value of one of the string length variables `lenX` (line 8) or `lenY` (line 12).

The combination of the padding character and the reifications eases the writing of safe models that make use of the string indexing operation, which, as discussed in Chapter 10, is a partial function. It is still possible to write an unsafe padded string model, although such a model could then be transformed into a safe model using the methodology described in [35].

Padded-String Implementation

Using the padded-string method requires no additional implementation; rather, the encoding of strings into integer-variable arrays occurs wholly at the modeling level. The required decomposition of a string constraint can be adequately accomplished via macro expansion, in the spirit of the flattening of `MINIZINC` models [72], or, indeed, by defining predicates in `MINIZINC` itself.

The Iterative Method

There is an alternate method for representing bounded-length strings as fixed-length sequences, not evaluated here. In the *iterative method* the length of each sequence is initially fixed to a lower bound; if no set of satisfying strings can be found, then the vector of lengths is increased lexicographically and search is restarted. Modeling string CSPs is significantly simpler when using the iterative method than when using the padded string method, and ensuring the safety of the resulting models is also more straightforward in the former than the latter. However, this ease of modeling comes with a substantial performance cost: models written using the iterative method perform consistently worse than those written using the padded-string method (see [VI]).

11.1.2 Search

A naïve heuristic for branching on string variables would consider in isolation the sets of candidate characters and the set of candidate lengths for each string variable, and create decisions using some standard value selection heuristic for integer variables. Momentary consideration shows that such naïve branching heuristics for string variables are not very compelling. It may be reasonable to partition a set of candidate lengths by splitting it at its median value, or by selecting its greatest or least element, but for a set of candidate characters these options are not very interesting. Clearly the proper design of branching heuristics for string variables, whether of bounded or unbounded length, is an interesting topic for future research, but is beyond the scope of this dissertation.

However, for the purpose at hand—namely, evaluating the efficiency of the bounded-length string variable implementations described in this thesis—it suffices to rely on a naïve branching heuristic for string variables. Different, less naïve branching heuristics should lead to lower runtimes, but it would make no difference to the *relative* runtimes of different string-variable type implementations.

In fact, there is a strong argument against using a well-designed string branching heuristic here. A proper comparison of runtimes between the six implementations listed in Section 11.1.1 requires using the same branching heuristic for every implementation. As two of those six implementations represent each string as an aggregation of integer variables, our choice of branching heuristics is limited to those that can be implemented on those representations. The

simplest strategy, which we have followed, is to implement a naïve branching heuristic for string variables that mimics a combination of the variable and value selection heuristics for integer variables included with GECODE. Thus, the possible value selection heuristics are restricted to those in which sets of candidate lengths and sets of candidate characters are treated the same, as each of these sets is represented by an integer variable in the aggregate-string and padded-string methods.

For our tests, we used two naïve value selection heuristics. Both heuristics create binary choice points, i.e., the domain is partitioned into exactly two parts. In the first heuristic, `cmin-lmin`, the two partitions either fix a component to its minimum value, or exclude that minimum value. So, for example, when branching on a set of candidate lengths for a string variable, the first decision restricts candidate strings to be as short as possible, and the second decision removes the shortest candidate length. In the second heuristic, `cmed-lmed`, a component is partitioned at its median element. So, for example, when branching on a set of candidate lengths for a string variable, the first decision considers the lower half of the set of candidate lengths, and the second decision considers the upper half of that set.

The heuristic for selecting a character to branch on should treat sets of candidate characters at mandatory and optional indices differently; sets of candidate characters at forbidden indices are already the empty set. Intuitively, branching on characters at optional indices serves little purpose; the branching heuristic should either select a set of candidate characters at a mandatory index, or branch on the set of candidate lengths (thereby creating more mandatory indices for at least one of the two decisions).

Our naïve branching heuristic, therefore, is as follows: if some mandatory indices exists, then branch over a set of candidate characters at such an index, else branch over the set of candidate lengths. To choose among sets of candidate characters at mandatory indices, both of our naïve heuristics choose a set of candidate characters of minimal cardinality. Note that both of these naïve heuristics are deterministic, deliberately so, as we wish to minimize potential differences in search among the various implementations in order to get a more accurate comparison of performance.

11.2 Benchmarks

Benchmark of KALUZA

The bounded-length string solver KALUZA [84] (see Section 7.3.3) includes over 50,000 instances that were generated for the symbolic execution of JavaScript, based on real-world Ajax web applications. The instances are written in an *ad hoc* constraint language¹ that can express constraints of equality, concat-

¹The benchmark of KALUZA is available at webblaze.cs.berkeley.edu/2010/kaluza/; translations of the benchmark into the SMTLIB2 language, using either the proposed

enation, length, and regular-language membership on string variables; while an integer variable may be constrained by a linear combination of constants, other integer variables, or string-variable lengths.

Unfortunately, these instances all turn out to be trivial for CP approaches (as already reported in [VI]), with runtimes below 0.01 seconds, and even the KALUZA runtimes are below half a second. In particular, the occurrence of regular-language membership constraints in the benchmark is limited to simple checks for the presence or absence of a constant substring. Hence this extensive benchmark is not particularly interesting. In order not to be biased by hand-picking among the 50,000 instances, we evaluated all the 14 instances that are distributed with the KALUZA code, removing those for which KALUZA generates incorrect or inconsistent results (as also reported in [106]).

Benchmark of NORN

A set of approximately 1,000 string constraint problems were generated for the *unbounded*-length string solver NORN [1].² These instances have regular-language membership constraints, generated by a model checker, based on counter-example-guided abstraction refinement (CEGAR), for string manipulating programs, as well as concatenation and length constraints on string variables, and linear constraints on integer variables. In practice, the benchmark of NORN differs from that of KALUZA primarily in the inclusion of more interesting regular-language membership constraints.

Instances of the benchmark of NORN are written in the cvc4 dialect of the SMTLIB2 language. These instances were translated into GECODE models using the three methods described in Section 11.1.1. The models were modified to remove unused variables: all instances in the benchmark appear to declare twelve string variables, but in the generated CP models any variable that does not participate in any constraint is discarded. Regular languages in the instances are specified as regular expressions; these were directly translated into GECODE's regular-expression language and modeled by $\text{REGULAR}_{\mathcal{L}}$ constraints, with the exception of expressions of the form $X \in \epsilon$, which were instead modeled by a constraint $\text{LEN}(X, 0)$. Approximately three quarters of all instances in the benchmark include a negated regular expression; as GECODE does not implement negation for regular expressions, these instances were omitted from the experiments. This decision was merely a matter of convenience; adding support for negating regular languages to GECODE is straightforward (e.g., [58]).

Note that we are solving the remaining 255 instances in a *bounded*-length context; our results are therefore incomparable with those of an unbounded-length solver such as NORN. For satisfiable instances NORN generates a language of all satisfying assignments for each string variable, whereas a CP-based method returns individual satisfying strings; furthermore, NORN can de-

string theory extension of cvc4 [60] or that of Z3-STR [106], are available at cvc4.cs.nyu.edu/papers/CAV2014-strings/.

²The benchmark of NORN is available at user.it.uu.se/~jarst116/norn/.

termine that an instance is unsatisfiable for strings of *any* length, whereas CP solvers are naturally limited to determining b -bounded unsatisfiability, for some practical upper bound b on string length. Nevertheless, the benchmark of `NORN` remains interesting in a *bounded*-length context, and is generally more challenging than the benchmark of `KALUZA`.

11.3 Experiments

Plotting Convention

Runtimes for the chosen benchmarks, shown in milliseconds and plotted with a logarithmic scale, are summarized in a series of Tukey boxplots [99], following the standard conventions: the bottom and top of each blue box are the first and third quartiles of the data, the red horizontal line inside the box is the second quartile (i.e., the median), the green square is the mean, and the ends of the whiskers are the lowest datum still within 1.5 times the inter-quartile range (IQR) of the lower quartile, and the highest datum still within 1.5 IQR of the upper quartile. Additionally, the actual data is overlaid as a set of partially transparent red dots with a small horizontal perturbation to reduce overstriking.

11.3.1 Effect of Block Size

The performance of both the **block list** and **block array** implementations could be dependent on the chosen size of the fixed-length blocks. The primary effect of block size is on the granularity of memory allocation: smaller block sizes lead to more frequent (de)allocation operations, while for larger block sizes a decrease to the upper bound of the set of candidate lengths will less frequently lead to freed memory. For the block list, block size also affects the worst-case time complexity of lookup operations, as the time to locate a particular set of candidates is linear in the number of blocks in the list.

To evaluate the effect of block size on overall performance, the 255 instances of the benchmark of `NORN` were run for both the block list and block array implementations, using blocks of 4, 16, and 64 bitsets for each. Each instance was tried for maximum string lengths $b \in \{16, 32, 64, 128\}$. Figures 11.2 to 11.4 summarize the results.

Figure 11.2 shows the time needed to find the first solution or prove none exists. The effect of block size is clearly negligible in this experiment, the only observable effect being for very short strings ($b = 16$), for which block size 4 performs slightly worse than the other choices. This pattern is repeated in Figures 11.3 and 11.4, which show the time to find up to the first 1,000 solutions or prove none exist, using either the **cmin-lmin** branching heuristic (Figure 11.3) or the **cmed-lmed** branching heuristic (Figure 11.4).

It is apparent, for this benchmark at least, that the effect that the choice of block size has on the performance of these two implementations is insignificant

Figure 11.2. Comparison of runtime (in milliseconds) to find the first solution or prove none exists, using the `cmin-lmin` branching heuristic, on the benchmark of `NORN`. The benchmark is evaluated for b -length sequences implemented as either a list of blocks (left column) or an array of pointers to blocks (right column), for maximum string lengths $b \in \{16, 32, 64, 128\}$ (rows, top to bottom) and block sizes 4, 16, and 64 (left to right in each figure).

Figure 11.3. Comparison of runtime (in milliseconds) to find up to the first 1,000 solutions or prove none exists, using the `cmin-lmin` branching heuristic, on the benchmark of `NORN`. The benchmark is evaluated for b -length sequences implemented as either a list of blocks (left column) or an array of pointers to blocks (right column), for maximum string lengths $b \in \{16, 32, 64, 128\}$ (rows, top to bottom) and block sizes 4, 16, and 64 (left to right in each figure).

Figure 11.4. Comparison of runtime (in milliseconds) to find up to the first 1,000 solutions or prove none exists, using the `cmed-lmed` branching heuristic, on the benchmark of `NORN`. The benchmark is evaluated for b -length sequences implemented as either a list of blocks (left column) or an array of pointers to blocks (right column), for maximum string lengths $b \in \{16, 32, 64, 128\}$ (rows, top to bottom) and block sizes 4, 16, and 64 (left to right in each figure).

when compared to the runtimes required as the maximum string length b is increased. The problem appears to be that both the benchmark of `NORN` and the benchmark of `KALUZA` are quite solution-rich: many solutions have the same string lengths, thus there is little need for searching based on candidate string lengths. Determining the effect, if any, of block size on the performance of the **block list** and **block array** implementations will require a benchmark with a sparse solution space.

11.3.2 Runtime Comparisons

We now compare the runtime of the four variants of the string variable implementation (i.e., **block list**, **block array**, **dynamic array**, and **static array**) with that of the aggregate-string implementation (**aggregate**). As a control implementation, we further evaluated models written using the padded-string method (**padded**), which relies only upon the built-in integer variables of `GE-CODE`.

Results for the benchmark of `NORN` are shown in Figure 11.5 for maximum string lengths $b \in \{16, 32\}$, and Figure 11.6 for maximum string lengths $b \in \{64, 128\}$. Each instance was evaluated for the time to find the first solution or prove none exists, using the `cmin-lmin` branching method; and for the time to find up to 1,000 solutions or prove none exists, using both the `cmin-lmin` and `cmed-lmed` branching heuristics.

As expected in light of the results of Section 11.3.1, there is no observable difference between the **block list** and **block array** implementations. For small string lengths ($b = 16$), both of those implementations slightly outperform the **dynamic array** and **static array** implementations.

It should be noted that in both the **block list** and **block array** implementations, the *final* block in the data structure is sometimes smaller than the remaining blocks. The intent of this enhancement is obvious: to avoid allocating unnecessary storage for the sequence of sets of candidate characters, the total number of bitsets k in the data structure may be limited to the upper bound of the set $\mathcal{N}$ of candidate lengths at the time of the allocation. For a **block list** or **block array** where each block holds m bitsets, there would always be $m \cdot \left\lceil \frac{\max(\mathcal{N})}{m} \right\rceil \geq k$ bitsets allocated in total; by relaxing the block-size restriction on the final block only, the allocation of unnecessary bitsets can be avoided, with no increase to the complexity of the lookup operation. Without this minor enhancement, neither the **block list** nor **block array** implementations performs as well as either the **dynamic array** or **static array** implementation.

As for the remaining two implementations, both are generally slower than the native-string implementations by an order of magnitude. The **padded** implementation, already shown in [VI] to be slower than the **aggregate** implementation on all tested instances of the benchmark of `KALUZA`, fares no better

Figure 11.5. Comparison of runtime (in milliseconds) for several b -length sequence implementations for $b = 16$ (left column) and $b = 32$ (right column). Runtimes are for the following numbers of solutions and branching strategy: in the top row, runtime to find the first solution or prove none exists, using the `cmin-lmin` branching heuristic; in the middle row, runtime to find up to the first 1,000 solutions or prove none exists, using the `cmin-lmin` branching heuristic; and in the bottom row, runtime to find up to the first 1,000 solutions or prove none exists, using the `cmed-lmed` branching heuristic.

Figure 11.6. Comparison of runtime (in milliseconds) for several b -length sequence implementations for $b = 64$ (left column) and $b = 128$ (right column). Runtimes are for the following numbers of solutions and branching strategy: in the top row, runtime to find the first solution or prove none exists, using the *cmin-lmin* branching heuristic; in the middle row, runtime to find up to the first 1,000 solutions or prove none exists, using the *cmin-lmin* branching heuristic; and in the bottom row, runtime to find up to the first 1,000 solutions or prove none exists, using the *cmed-lmed* branching heuristic.

here on the combined benchmarks of `NORN` and `KALUZA`. For many instances that the native-string implementations finish in one second or less, the two other implementations time out.

Part III: Conclusion

Though I vowed to finish or die,
I am not yet dead,
and not quite finished.

TOM HOLBROOK

Conclusion

We have formalized an approximation of sets of bounded-length strings, the b -length sequence approximation, and demonstrated its suitability as a representation for the domain of a string variable in a constraint satisfaction problem. We have used this approximation as the formal basis of two different approaches to string variables in CP. In the first approach, we have translated our representation of a string variable into an aggregation of scalar integer variables that is directly implementable for any existing CP solver. In the second approach, we have designed and implemented a native string variable type for a CP solver.

Propagation algorithms for these two approaches are similar, as both approaches are based on the same bounded-length string representation. Starting from formal definitions of constraints over strings, we have derived, in a principled fashion, the strongest propagator functions possible on that representation. From these propagator functions, we have implemented propagator algorithms for both our representations.

Either of our implementations substantially eases the burden of modeling a string constraint problem—a non-trivial concern for string constraint problems in CP. Additionally, both implementations outperform the traditional method for modeling strings in CP, on some problems by orders of magnitude. Our custom string variable type clearly outperforms its aggregate cousin, though: through careful design, we were able to reduce the runtime complexity of common propagator operations, reduce memory requirement, and provide an easier modeling interface.

The design of our string variable type and its propagators has been greatly influenced by our prior development of a theoretical framework describing propagator strength via Galois connections. This framework provides for precise definitions of propagator strength even for complicated variables, such as our bounded-length strings. Basing our string representation on this framework enabled the propagator derivation mentioned above. It further served to clarify important properties of the representation as the implementation process progressed.

This framework, which we have presented in detail, is generally applicable to finite-domain CP solvers. From a simple central premise it helps us gain a deeper understanding of consistency and domain representations.

Future Work

Abstraction and Propagation

- $\langle\alpha, \gamma\rangle$ -consistency subsumes most of the well-known consistency definitions in CP, with the notable exception of bounds(D) consistency and range consistency. But a casual perusal of Figure 5.2 is sufficient to see that there exists some relationship to $\langle\alpha, \gamma\rangle$ -consistency. Possibly an extension of $\langle\alpha, \gamma\rangle$ -consistency to Galois connections on pre-orders would bring these two black sheep back into the fold. In any case, we could draw a similar figure for every $\langle\alpha, \gamma\rangle$ -consistency, and it would be interesting to explore the properties of the quasi- $\langle\alpha, \gamma\rangle$ -consistency found to each side.
- A related question is the relationship between $\langle\alpha, \gamma\rangle$ -consistency and $\phi\psi$ -completeness. Does $\phi\psi$ -completeness work for *any* combination of closure operators, or is there an implicit relationship between them?
- We have made the argument that $\langle\alpha, \gamma\rangle$ -consistency is a valuable tool for talking about structured types. Despite the fact that strings have very little in the way of structure, the representational consistency we based our string variable design upon turned out to be very useful. Are there other structured types that could benefit from a similar analysis?
- Finally, it would be interesting to expand the Galois-based framework presented here in order to formalize the relationship between representational and computational propagation strength, perhaps via a compositional definition with separate representation and computation phases. Ultimately, one could envision a sort of compositional algebra of consistency definitions, broad enough to embrace CP solvers that have eschewed the traditional constraint store.

Strings and Constraints

- Proper branching heuristics for string variables form an obvious direction of enquiry, with the potential for significant speed improvement. Our naïve branching heuristic has left plenty of room for improvement on this front.
- There are several advanced propagation techniques that seem promising for propagating string constraints, but it is not always obvious how to adapt them to the new type. Our early propagator designs assumed that views would be important for efficient propagation, but this has been surprisingly difficult to put into practice.
- We have defined one consistency for string domains, and since it is the basis of our design it is also the strongest consistency that our representation supports. But we have not investigated relaxations of this consistency. We could write propagators that consider only characters at mandatory indices, or in some window beyond a fixed prefix (similar to [47]), or possibly some other measure.

- Finally, as mentioned in Section 7.3.3, the quasi-unboundedness of the path variables in [47], while not sufficient for deciding unbounded-string problems, would nevertheless be an interesting extension to our implementation. Propagation over optional indices, when combined with a non-existent upper bound on length, makes it difficult to ensure termination. Perhaps one could define the relationship systematically, using something like a Galois connection...

Acknowledgments

It's hard to overstate my satisfaction.¹

GLADOS

Many people helped to make this research, and this monograph, possible. First and foremost, my advisors: Prof. Pierre Flener, Prof. Parosh Aziz Abdulla, and Justin Pearson. Thank you for giving me both the freedom to find a path and the encouragement to stay on it. I especially have to thank Pierre and Justin: I have learned so much from both of you—I cannot count the number of times that a major breakthrough started when Pierre caught me hiding sloppy thinking behind sloppy notation; or when Justin pointed out that my argument was just fine, except for being both backwards and wrong...

Many of the ideas in these pages were sparked by discussions with colleagues, at seminars, workshops, summer schools, and conferences. Particular thanks is due to Jean-Noël Monette, Guido Tack, Gustav Björdal, Thiago Costa Porto, Mats Carlsson, and above all, Prof. Christian Schulte, for help both practical and theoretic.

Writing this dissertation has been non-trivial. Besides my advisors, I also want to thank Jennifer Ast for her dogged insistence that someone ought to be able to *read* the first part; Camilla Eriksson from Thesis Production for helping make this into an actual thing; and a special thanks to Sofia Cassel for turning the introduction into the sammanfattning, somehow making the introduction better in the process—for a Canadian, your Swedish is pretty impressive.

Being a PhD student is not all research—as a teaching assistant, I had the good luck to work for Pierre Flener and Olle Gällmo, two people whose teaching inspired me as a Masters student here at Uppsala, and continued to inspire me as a PhD student. I have had many excellent teachers along the long road that has been my formal education; now that it draws to an end, it's high time that I thanked some of them. In roughly reverse chronological order (from my viewpoint): Benjamin Keller, Gisela Ahlbrandt, and Matthew Evett (Eastern Michigan University); Bill Reichert (Washtenaw Community College); Phyllis Mentzell Ryder (University of Arizona); William Wimsatt and Noel Swerdlow (University of Chicago); Susan Blanchett (Henry Clay High School); Paul Sprague (Morton Jr. High School); and Mrs. Campbell, my third grade math teacher at Brewton Elementary. A sincere thanks to you all: I would not have gotten here without you.

¹If you read the whole section, then there will be cake. Honest.

And Jennifer. Where do I begin? You are my best friend, the only person I always want to see, the love of my life. You totally kick ass. I wouldn't have done this without your constant support and encouragement. I look forward to seeing you at dinner again now that this is done.

My family has been a never ending source of support. Thanks to my parents, Tim and Susie, who always believed I could do this; my sisters, Soren and Molly (who probably didn't); my niece, Mia, who thinks it's cool that I'm a robot doctor now; my Michigan(ish) family, who have always done me the immense honor of treating me like one of the cousins; and last, but certainly not least, Lionel, Joëlle, Nora and Sanna, who have done more to keep my sanity intact than they know.

To my fellow PhD students (and Post Doc) in the ASTRA group, Jun, Farshid, Andreína, Loïc, and Jean-Noël: you guys have been there for me through this whole experience. I don't think I would have made it without your help, distraction, cookies, commiseration, friendship, and cookies. Thanks also to my fellow teaching assistants, Nikos, Pontus, Kjell, and Peter: teaching wouldn't have been nearly as fun without you. Andreína and Sofia deserve special thanks for insisting that lunch is important, and for generally being better friends than I deserve. I'd also like to thank Linnéa and the crew from SwAc; the members of the Uppsala ACM-w chapter; Virginia; Lena and Emma and all the gang; Yunyun and Edith; Patricia Limerick; and Mo and Jesse.

Finally, my friends on distant shores to whom I have been like a ghost these last few years, to you I say: This! This is what I've been doing. Sigh. But look at me, still talking when there's science to do.³

About the Cover

The cover is an abstract representation of the abstraction illustrated in Figure 2.3(c), drawn in the abstract style of Hilma av Klint (1862–1944), a Swedish artist and theosophist who painted her first abstract compositions several years before Kandinsky invented the idea. The artwork was kindly provided by the ever talented Soren Scott, an itinerant research coder who spends her days in Boulder, Colorado, and who is not, to the best of my knowledge, a theosophist.

³Incidentally, there's bad news about that cake...

My Publications

- [I] Roger Kameugne, Laure Pauline Fotso, Joseph Scott, and Youcheu Ngo-Kateu.
“A Quadratic Edge-Finding Filtering Algorithm for Cumulative Resource Constraints”. In: *Principles and Practice of Constraint Programming (CP 2011)*. Proceedings of the 17th International Conference. (Perugia, Italy, Sept. 12–16, 2011). Edited by Jimmy Ho-Man Lee. LNCS 6876. Springer, 2011, pages 478–492.
- [II] Roger Kameugne, Laure Pauline Fotso, and Joseph Scott.
“A Quadratic Extended Edge-Finding Filtering Algorithm for Cumulative Resource Constraints”. In: *International Journal of Planning and Scheduling* 1.4 (2013), pages 264–284.
- [III] Joseph Scott.
“Rapid Prototyping of a Structured Domain through Indexical Compilation”. In: *COSpeL: The 1st Workshop on Domain Specific Languages in Combinatorial Optimization*. (Uppsala, Sweden, Sept. 16, 2013). Satellite workshop of CP 2013. 2013.
Available from: cp2013.a4cp.org/workshops/cospeL.
Cited on pages 110, 129.
- [IV] Joseph Scott, Pierre Flener, and Justin Pearson.
“Bounded Strings for Constraint Programming”. In: *Tools with Artificial Intelligence (ICTAI 2013)*. Special Track on SAT and CSP technologies. (Washington, D.C., U.S.A. Nov. 4–6, 2013). IEEE Computer Society, 2013, pages 1036–1043.
Cited on pages 53, 108, 110.
- [V] Roger Kameugne, Laure Pauline Fotso, Joseph Scott, and Youcheu Ngo-Kateu.
“A Quadratic Edge-Finding Filtering Algorithm for Cumulative Resource Constraints”. In: *Constraints* 19.3 (2014), pages 243–269.
Extended version of [I].
- [VI] Joseph Scott, Pierre Flener, and Justin Pearson.
“Constraint Solving on Bounded String Variables”. In: *Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015)*. Proceedings of the 12th International Conference. (Barcelona, Spain, May 18–22, 2015). Edited by Laurent Michel. LNCS 9075. Springer, 2015, pages 375–392.
Cited on pages 24, 25, 108–110, 129, 171, 173, 178.

Sammanfattning på svenska

Ich entschuldige, dass ich nicht in Ihrer
sondern in meiner Sprache geschrieben.
So lieb mir auch die erste ist, so schreibe
ich doch zu schlecht in derselben, es war
nötig, dass ich mich über alles bestimmt
ausdrücke und Ewr Hochedelgeb. lesen die
Werke meine Nation ohne Anstoss.¹

JOHANN GOTTLOB GEORGI,
writing to Torbern Bergman,
Professor of Chemistry,
Uppsala University,
9 August 1768

Villkorsprogrammering baseras på koncept som torde vara bekanta för alla som någon gång försökt lösa ett sudoku (se Figur 11.7). *Sudoku* är ett slags sifferpussel eller logikbaserat spel: ett 9×9 rutnät är indelat i fyra regioner med vardera 3×3 rutor. För att lösa ett sudoku placerar man ett heltal mellan 1 och 9 i var och en av de totalt 81 rutorna. Varje heltal får förekomma endast en gång i varje rad, kolumn, respektive region. Ett sudoku har alltid ett antal heltal utplacerade i förväg, s.k. *ledtrådar*.

Sudoku är ett *kombinatoriskt problem*. Ett sådant problem har ett mycket stort (men inte oändligt) antal möjliga lösningar, och det går att avgöra vilka lösningar som är korrekta resp. inkorrekta genom att följa olika regler. I sudoku, liksom många andra kombinatoriska problem, finns det dock alldeles för många möjliga lösningar för att det ska vara idé att försöka hitta en som är korrekt genom 'trial and error'-metoden. De regler som kan användas för att avgöra om en lösning är korrekt går heller inte på ett uppenbart sätt att använda för att räkna ut en lösning.

Ett sätt att handskas med kombinatoriska problem är att se dem som *villkorsproblem*. Ett sudoku kan uttryckas som följande villkorsproblem:

- Vi kallar varje okänt värde (d.v.s., var och en av de 81 rutorna) en *variabel*.

¹"I apologize that I write to you not in your language but rather in mine. As dear to me as the former is, I do however write so poorly in it, [and] it was necessary that I express myself clearly about everything and Your Lordship reads the works of my country without offense." (Gordon, Michael D. *Scientific Babel*. [Chicago: University of Chicago Press, 2015], 45)

9		6		7		4		3		9		6		7		4	<div>2 3 4 6 8 9</div>	3	
			4			2							4			2	<div>2 3 4 6 8 9</div>		
	7			2	3		1				7			2	3		<div>1</div>		
5						1				5						1	<div>2 3 4 6 8 9</div>		
	4		2		8		6				4		2		8		<div>6</div>		
		3						5				3					<div>2 3 4 6 8 9</div>	5	
	3		7				5				3		7				<div>5</div>		
		7			5							7			5		<div>2 3 4 6 8 9</div>		
4		5		1		7		8		4		5		1		7	<div>2 3 4 6 8 9</div>	8	

(a) Sudoku med 29 ledtrådar (b) ALLDIFFERENT (kolumn 8)

Figur 11.7. Ett sudoku (a), och ett av de 27 villkoren för sudokut (b).

- Varje variabel har ett antal möjliga värden (i sudoku, heltalen $\{1, \dots, 9\}$). Dessa värden kallas variabelns *domän*.
- Vi kan skilja på möjliga och omöjliga lösningar med hjälp av *villkor* som bestämmer vilka kombinationer av värden de olika variablerna får ha.

I sudoku kan de tre punkterna, eller delvillkoren, ovan uttryckas som 27 villkor, eftersom varje delvillkor kan appliceras på nio kombinationer av nio variabler. Varje delvillkor definierar en grupp av variabler som alla måste anta olika värden. Sudoku kan därmed sägas vara uppbyggt av 27 instanser av samma *kombinatoriska substruktur* för olika kombinationer av variabler. Just den här substrukturen kallas ofta för ALLDIFFERENT-villkoret, men det finns många andra villkor som tillsammans kan användas för att lösa olika villkorsproblem. Ett villkorsproblem kan beskrivas som en konjunktion av olika villkor, och kallas då en *villkorsmodell*. Figur 11.8 visar en villkorsmodell av sudoku. Modellen är egentligen bara en omskrivning av reglerna för sudoku så att de uttrycks som ALLDIFFERENT-villkor istället.

Villkor är dock inte bara smarta sätt att beskriva olika kombinatoriska problem: inom *villkorsprogrammering* är de centrala för att kunna lösa problemen. Med villkorsprogrammering kan vårt sudoku lösas ungefär så här: Börja med att förutsätta att alla rutor kan innehålla alla heltal (förutom de rutor som redan innehåller ett bestämt heltal). Ta sedan en regel i taget, och utesluter de heltal som regeln inte tillåter. Till exempel kan man börja med regeln “varje heltal får förekomma endast en gång i kolumn 8”. Figur 11.7b visar vad som händer när man applicerar denna regel. Tre rutor i kolumn 8 har redan ett bestämt heltal (1, 5, resp. 6) så dessa heltal kan uteslutas från övriga rutor i kolumnen. I figuren är uteslutna heltal markerade i grått. En procedur som utesluter olika möjliga värden på det här sättet, genom att kontrollera olika villkor, kal-


```

1 array[1..9,1..9] of var 1..9: puzzle;
2
3 constraint % All different in rows:
4   forall (i in 1..9) (
5     alldifferent( [ puzzle[i,j] | j in 1..9 ] ) );
6
7 constraint % All different in columns:
8   forall (j in 1..9) (
9     alldifferent( [ puzzle[i,j] | i in 1..9 ] ) );
10
11 constraint % All different in sub-squares:
12   forall (a, o in 1..3) (
13     alldifferent( [ puzzle[(a-1)*3 + a1, (o-1)*3 + o1] |
14       a1, o1 in 1..3 ] ) );

```

Figur 11.8. Modell av sudoku, formulerat i modelleringsspråket MINIZINC. Taget från [66].

las inom villkorsprogrammering för en *propagator*. Ett program som kan lösa villkorsproblem kallas ofta för en *lösare* (eng. 'solver').

Även om sudoku-exemplet kan ses som ett 'leksakerproblem' betyder inte det att villkorsprogrammering bara kan användas för att lösa olika logikspel och pussel. Samma principer används för att lösa större och mer komplexa problem i verkligheten.

Villkorsprogrammering med strängar

En *sträng* är i detta sammanhang en ändlig sekvens av tecken som kommer från ett *alfabet*. Till exempel är 'plantskolan' och 'nödtvång' strängar från alfabetet

$$\{a, b, c, d, \dots, z, \text{å}, \text{ä}, \text{ö}\}$$

Det finns naturligtvis andra möjliga alfabet, t.ex. sådana som innehåller heltal eller olika tecken för interpunktion.

Villkorsproblem kan innehålla *strängvariabler*, d.v.s., variabler vars domän är en mängd strängar. Strängvariabler lyder under strängvillkor, och strängvillkor finns i en stor mängd olika verkliga problem såsom testfallsgenerering, programanalys och websäkerhet.

Under senare år har det forskats mycket på att lösa villkorsproblem som innehåller strängvariabler. Det finns i princip tre olika sätt att behandla strängvariabler inom villkorsprogrammering (och inom datavetenskap rent generellt). Vi illustrerar dem genom att visa hur strängvariabler kan implementeras i det klassiska programmeringsspråket ADA (men principerna är de samma inom villkorsprogrammering):

- som en sträng med *bestämd* längd (kan endast representera värden av en särskild längd),

- *begränsad* längd (kan endast representera värden som är högst så långa som begränsningen anger), eller
- *obegränsad* längd (kan representera värden med vilken längd som helst, så länge inte minnet tar slut).

Inom villkorsprogrammering kan en sträng med bestämd längd (n) representeras som en sekvens av n variabler. Den första variabeln representerar första tecknet i strängen, och variabelns domän är strängens alfabet. Det kan hända att den lösare för villkorsproblem man använder inte har stöd för variabler som kan ta värden från det alfabet man vill använda, men då kan man istället koppla alfabetet till heltal, så att a representeras som 1, b som 2, etc. Den främsta fördelen med detta är att man kan använda existerande lösare för villkorsprogrammering istället för att behöva skriva en egen för varje problem som innehåller strängar. Det finns nämligen ett stort antal villkor som redan är identifierade och beskrivna (t.ex. `ALLDIFFERENT` som vi nämnde ovan), och lösare för villkorsprogrammering har färdigimplementerade metoder (propagatorer) för många av dessa villkor. Nackdelen med att modellera strängar på det här sättet är att man får en mycket snävare tolkning av villkorsproblemet när man på förhand måste bestämma hur många variabler man behöver använda för att representera varje sträng. Det är inte heller möjligt att representera en sträng med obegränsad längd som en sekvens av (heltals)variabler, eftersom sådana har en ändlig domän i lösare för villkorsprogrammering. Man kan däremot välja att lägga till en ny variabeltyp för strängar med obegränsad längd.

En lösare som har en strängvariabeltyp och propagatorer för strängvillkor gör det möjligt att modellera villkorsproblem med strängar mycket mer kompakt och begripligt. Varje sträng i villkorsproblemet representeras av en enskild strängvariabel, och strängvillkoren kan uttryckas direkt. Genom att reducera antalet variabler och villkor i en modell kan det också bli lättare för lösaren att hitta en korrekt lösning. Det krävs dock en del arbete för att utveckla och implementera en ny variabeltyp, propagatorer, och lösningsmetoder för lösaren, så man måste göra en avvägning.

Strängvariabeltyper som tillåter strängar av obegränsad längd kan ha en oändlig domän. För att representera en sådan domän mer elegant och kompakt, kan man välja att specificera den i termer av egenskaper hos strängarna, istället för att försöka räkna upp alla strängar den innehåller. En domän med strängar som innehåller exakt ett a följt av ett godtyckligt antal b , är t.ex. oändligt stor. Den kan då representeras genom att skriva ab^* , vilket beskriver hela domänen i termer av en egenskap som delas av alla strängar i den. ab^* kallas ett *reguljärt uttryck*.

Tyvärr visar det sig att kompakta representationer som t.ex. reguljära uttryck medför en prestandaförlust, förutom arbetet som krävs för att implementera en ny variabeltyp; denna förlust är relaterad till propagatorer. Propagatorer tar reda på vilka värden som kan uteslutas ur en variabls domän, och sådana procedurer blir alldeles för dyra (i termer av tidskomplexitet) när man använder kompakta domänrepresentationer. Strängvariabeltyper med obegränsad längd

går därför inte i nuläget att använda i praktiken i lösare för villkorsprogrammering.

Vi tror att fördelarna med att utöka lösare för villkorsprogrammering med en strängvariabeltyp överväger nackdelarna. I denna avhandling arbetar vi dock med lösare vars variabler har ändliga domäner. En variabeltyp som bara tillåter strängar med bestämd längd är inte lockande (enligt diskussionen ovan), och en variabeltyp som tillåter strängar med obegränsad längd är inte realistiskt. Vi har därför valt att implementera en variabeltyp som tillåter strängar med *begränsad* längd.

Galoiskopplingar och propagering

Valet av domänrepresentation för en strängvariabel med begränsad längd beror mycket på hur variabeln ska interagera med propagatorer. En central fråga är hur starka propagatorerna ska vara: vad ska de resonera om, och vilken förväntad effekt ska de ha på variabelns domän? Inom villkorsprogrammering svarar man på dessa två delfrågor i termer av *fullständighet* (eng. 'completeness') och *konsistens* (eng. 'consistency'). Fullständighet är en egenskap hos en propagator, medan konsistens är en egenskap hos en domän.

Det visar sig vara svårt att definiera fullständighet och konsistens på ett tillfredsställande sätt för strängvariabler med begränsad längd. En propagator kan ses som en beräkning av ett villkor på en abstraktion av en variabels domän. Propagatorer utesluter värden som aldrig kan finnas med i en lösning som lyder villkoret: den starkaste tänkbara propagatorn skulle utesluta alla sådana värden, men de flesta propagatorer är svagare och utesluter bara något eller några sådana värden. Man kan tänka sig att en svag propagator använder sig av en förenklad representation av den faktiska domänen. En svag propagator gör först en approximation av den faktiska domänen genom att ta bort detaljer. Det gör att propagatorns beräkningar blir enklare och snabbare. Sedan omvandlas approximationen tillbaka till den faktiska domänen, men då med något sämre precision: vissa värden som skulle ha kunnat uteslutas från den faktiska domänen finns kvar, eftersom beräkningen gjordes på en approximation av domänen istället.

För att en propagering ska bli korrekt måste de omvandlingar propagatorn gör följa vissa regler. Det visar sig dock att den faktiska domänen och propagatorns approximation ofta är relaterade till varandra genom en Galoiskoppling. Galoiskopplingar har den trevliga egenskapen att de är enkla att definiera men har långtgående effekter.

För att definiera fullständighet och konsistens för strängvariabler med begränsad längd utvecklar vi i denna avhandling ett teoretiskt ramverk för propagatorers styrka, baserat på Galoiskopplingar. Detta ramverk visar sig vara mycket användbart: Till exempel är strängar av begränsad längd just så komplicerade att det inte alltid är uppenbart hur variabler och propagatorer ska

interagera – tills man tar hänsyn till den Galoiskoppling som definierar domänenrepresentationen.

Slutsatser

Vi har använt vårt teoretiska ramverk som en formell bas för två olika metoder att hantera strängvariabler med begränsad längd inom villkorsprogrammering. I den första metoden översätter vi vår strängvariabelrepresentation till en samling heltalsvariabler som direkt kan implementeras av en lösare för villkorsprogrammering. I den andra metoden har vi utvecklat och implementerat en särskild strängvariabeltyp istället.

Vi har även implementerat propagatorer för strängvariabler baserat på båda metoderna. Båda dessa implementationer gör det enklare att modellera problem med strängvillkor, vilket är icke-trivialt inom villkorsprogrammering. Båda presterar också bättre (och ibland mycket bättre) än den traditionella metoden för att modellera strängar inom villkorsprogrammering. Vår strängvariabeltyp presterar dessutom bättre än samlingsrepresentationen, eftersom vi lyckades utveckla den så att propagatorerna kunde bli mer effektiva, använda mindre minne, och vara enklare att modellera. Det är alltså svårt att se några nackdelar med strängvariabler med begränsad längd.

Genom att vårt ramverk baseras på Galoiskopplingar blev det enklare att utveckla propagatorer, och också att definiera viktiga egenskaper hos representationerna under tiden strängvariablerna implementerades. Vårt ramverk är generellt applicerbart på lösare för villkorsprogrammering som använder variabler med ändliga domäner. Det ger oss också en djupare förståelse för konsistens och för olika sätt att representera domäner.

Bibliography

- [1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman.
“String Constraints for Verification”. In: *Computer Aided Verification (CAV 2014)*. Proceedings of the 26th International Conference. (Vienna, Austria, July 18–22, 2014). Edited by Armin Biere and Roderick Bloem. LNCS 8559. Springer, 2014, pages 150–166.
Cited on pages 103, 105, 106, 173.
- [2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman.
“Norn: An SMT Solver for String Constraints”. In: *Computer Aided Verification (CAV 2015)*. Proceedings of the 27th International Conference, Part I. (San Francisco, CA, USA, July 18–24, 2015). Edited by Daniel Kroening and Corina S. Pasareanu. LNCS 9206. Springer, 2015, pages 462–469.
Cited on pages 105, 106.
- [3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri.
“OpenFST: A General and Efficient Weighted Finite-State Transducer Library”. In: *Implementation and Application of Automata (CIAA 2007)*. Revised Selected Papers of the 12th International Conference. (Prague, Czech Republic, July 16–18, 2007). Edited by Jan Holub and Jan Zdárek. LNCS 4783. Springer, 2007, pages 11–23. Library available from openfst.org.
Cited on page 168.
- [4] Henrik Reif Andersen, Tarik Hadzic, John N. Hooker, and Peter Tiedemann.
“A Constraint Store Based on Multivalued Decision Diagrams”. In: *Principles and Practice of Constraint Programming (CP 2007)*. Proceedings of the 13th International Conference. (Providence, RI, USA, Sept. 23–27, 2007). Edited by Christian Bessière. LNCS 4741. Springer, 2007, pages 118–132.
Cited on page 28.
- [5] Krzysztof R. Apt.
“The Essence of Constraint Propagation”. In: *Theoretical computer science* 221.1-2 (1999), pages 179–210.
Cited on page 50.
- [6] F. Azevedo.
“Cardinal: A finite sets constraint solver”. In: *Constraints* 12.1 (2007), pages 93–129.
Cited on page 95.
- [7] Francisco Azevedo.
“Constraint Solving Over Multi-Valued Logics: Application to Digital Circuits”. PhD thesis. Universidade Nova de Lisboa, 2002.
Cited on page 95.
- [8] Roman Barták.
“Dynamic global constraints in backtracking based environments”. In: *Annals of Operations Research* 118.1 (2003), pages 101–119.
Cited on page 107.

- [9] Mohammed Said Belaid, Claude Michel, and Michel Rueher.
 “Boosting Local Consistency Algorithms over Floating-Point Numbers”.
 In: *Principles and Practice of Constraint Programming (CP 2012)*. Proceedings of
 the 18th International Conference. (Québec City, QC, Canada, Oct. 8–12, 2012).
 Edited by Michela Milano. LNCS 7514. Springer, 2012, pages 127–140.
 Cited on page 50.
- [10] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit.
 “Global Constraint Catalogue: Past, Present, and Future”. In: *Constraints* 12.1 (Mar.
 2007). The catalogue is at <http://sofdem.github.io/gccat>, pages 21–62.
 Cited on pages 15, 27.
- [11] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit.
 “Deriving Filtering Algorithms from Constraint Checkers”.
 In: *Principles and Practice of Constraint Programming (CP 2004)*. Proceedings of
 the 10th International Conference. (Toronto, Canada, Sept. 27–Oct. 1, 2004).
 Edited by Mark Wallace. LNCS 3258. Springer, 2004, pages 107–122.
 Cited on page 103.
- [12] Frédéric Benhamou.
 “Heterogeneous Constraint Solving”. In: *Algebraic and Logic Programming*
 (ALP 1996). Proceedings of the 5th International Conference.
 (Aachen, Germany, Sept. 25–27, 1996).
 Edited by Michael Hanus and Mario Rodríguez-Artalejo. LNCS 1139. Springer, 1996,
 pages 62–76.
 Cited on pages 47, 50–52.
- [13] Christian Bessière.
 “Constraint Propagation”. In: *Handbook of Constraint Programming*.
 Edited by Francesca Rossi, Peter van Beek, and Toby Walsh.
 Amsterdam: Elsevier, 2006. Chapter 3, pages 29–83.
 Cited on pages 27, 31, 46, 47, 50.
- [14] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh.
 “Disjoint, Partition and Intersection Constraints for Set and Multiset Variables”.
 In: *Principles and Practice of Constraint Programming (CP 2004)*. Proceedings of
 the 10th International Conference. (Toronto, Canada, Sept. 27–Oct. 1, 2004).
 Edited by Mark Wallace. LNCS 3258. Springer, 2004, pages 138–152.
 Cited on pages 54, 77, 89.
- [15] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan.
 “WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering
 Exploit Construction”. In: *Computer and Communications Security (CCS 2011)*.
 Proceedings of the 18th ACM Conference.
 (Chicago, Illinois, USA, Oct. 17–21, 2011).
 Edited by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM, 2011,
 pages 575–586.
 Cited on page 95.
- [16] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov.
 “Path Feasibility Analysis for String-Manipulating Programs”.
 In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*.
 Proceedings of the 15th International Conference. (York, UK, Mar. 22–29, 2009).
 Edited by Stefan Kowalewski and Anna Philippou. LNCS 5505. Springer, 2009,
 pages 307–321.
 Cited on pages 95, 104, 107.

- [17] J Richard Büchi and Steven Senger.
 “Definability in the existential theory of concatenation and undecidable extensions of this theory”. In: *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 34 (1988), pages 337–342.
 Cited on page 104. Reprinted in: J. Richard Büchi.
The Collected Works of J. Richard Büchi. Edited by Saunders Mac Lane and Dirk Siefkes. New York: Springer, 1990.
- [18] Mats Carlsson.
SICStus Prolog User’s Manual 4.3. Edited by Thom Fruehwirth.
 available from <https://sicstus.sics.se/documentation.html>.
 Books on Demand, 2014. 412 pages.
 Cited on page 91.
- [19] Mats Carlsson, G. Ottosson, and B. Carlson.
 “An Open-Ended Finite Domain Constraint Solver”.
 In: *Programming Languages: Implementations, Logics, and Programs (PLILP 1997)*.
 Proceedings of the 9th International Symposium.
 (Southampton, UK, Sept. 3–5, 1997).
 Edited by Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen. LNCS 1292.
 Springer, 1997, pages 191–206.
 Cited on pages 91, 129.
- [20] Yves Caseau.
 “Abstract Interpretation of Constraints on Order-Sorted Domains”.
 In: *International Symposium on Logic Programming (ISLP 1991)*. Proceedings.
 (San Diego, California, USA, Oct. 28–Nov. 1, 1991).
 Edited by Vijay A. Saraswat and Kazunori Ueda. MIT Press, 1991, pages 435–452.
 Cited on page 50.
- [21] Chiu Wo Choi, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J. Stuckey.
 “Finite Domain Bounds Consistency Revisited”.
 In: *Advances in Artificial Intelligence (AI 2006)*. 2006-12-04/2006-12-08.
 (Proceedings of the 19th Australian Joint Conference on Artificial Intelligence).
 Edited by Abdul Sattar and Byeong Ho Kang. LNCS 4304. Springer, 2006,
 pages 49–58.
 Cited on pages 51, 52, 80, 83–85.
- [22] Philippe Codognet and Daniel Diaz.
 “Compiling Constraints in clp(FD)”.
 In: *The Journal of Logic Programming* 27.3 (1996), pages 185–226.
 Cited on page 129.
- [23] Hélène Collavizza and Michel Rueher.
 “Exploration of the Capabilities of Constraint Programming for Software Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*. Proceedings of the 12th International Conference.
 (Vienna, Austria, Mar. 25–Apr. 2, 2006).
 Edited by Holger Hermanns and Jens Palsberg. LNCS 3920. Springer, 2006,
 pages 182–196.
 Cited on page 95.
- [24] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck.
 “CPBPV: a constraint-programming framework for bounded program verification”.
 In: *Constraints* 15.2 (2010), pages 238–264.
 Cited on page 95.

- [25] Marco Correia.
 “Modern Techniques for Constraint Solving: The CaSPER Experience”.
 PhD thesis. Universidade Nova de Lisboa, Portugal, 2010.
 Cited on pages 50–52.
- [26] Marco Correia and Pedro Barahona.
 “View-based Propagation of Decomposable Constraints”.
 In: *Constraints* 18.4 (2013), pages 579–608.
 Cited on pages 47, 50–52, 86, 88.
- [27] Patrick Cousot and Radhia Cousot.
 “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Principles of Programming Languages (POPL 1977)*. Conference Record of the Fourth ACM Symposium on. (Los Angeles, California, USA, Jan. 1977).
 Edited by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pages 238–252.
 Cited on page 50.
- [28] Patrick Cousot and Radhia Cousot.
 “Systematic Design of Program Analysis Frameworks”.
 In: *Principles of Programming Languages (POPL 1979)*, Sixth Annual ACM Symposium on, Conference Record. (San Antonio, Texas, USA, Jan. 1979).
 Edited by Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen. ACM Press, 1979, pages 269–282.
 Cited on page 43.
- [29] B. A. Davey and H. A. Priestley.
Introduction to Lattices and Order. 2nd edition.
 Cambridge: Cambridge University Press, 2002.
 Cited on page 37.
- [30] Martin Davis, Ron Sigal, and Elaine J Weyuker.
Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science. San Diego: Academic Press, 1994.
 Cited on page 95.
- [31] Rina Dechter.
Constraint Processing. San Francisco: Morgan Kaufmann, 2003.
 Cited on pages 52, 83, 85.
- [32] Grégoire Dooks.
 “The CP(Graph) Computation Domain in Constraint Programming”.
 PhD thesis. Université Catholique de Louvain, Belgium, 2006.
 Cited on page 136.
- [33] Ingvar Eidhammer, Inge Jonassen, Sverre Helge Grindhaug, David Gilbert, and Madu Ratnayake.
 “A Constraint Based Structure Description Language for Biosequences”.
 In: *Constraints* 6.2/3 (2001), pages 173–200.
 Cited on page 107.

- [34] Michael Emmi, Rupak Majumdar, and Koushik Sen.
 “Dynamic Test Input Generation for Database Applications”. In: *Software Testing and Analysis (ISSTA 2007)*. Proceedings of the ACM/SIGSOFT International Symposium. (London, UK, July 9–12, 2007).
 Edited by David S. Rosenblum and Sebastian G. Elbaum. ACM, 2007, pages 151–162.
 Cited on pages 95, 105.
- [35] Alan M Frisch and Peter J Stuckey.
 “The proper treatment of undefinedness in constraint languages”.
 In: *Principles and Practice of Constraint Programming (CP 2009)*. Proceedings of the 15th International Conference. (Lisbon, Portugal, Sept. 20–24, 2009).
 Edited by Ian P. Gent. LNCS 5732. Springer, 2009, pages 367–382.
 Cited on pages 149, 154, 170.
- [36] Xiang Fu, Michael C. Powell, Michael Bantegui, and Chung-Chih Li.
 “Simple linear string constraints”.
 In: *Formal Aspects of Computing* 25 (Nov. 2013), pages 847–891.
 SUSHI is available from http://people.hofstra.edu/Xiang_Fu/XiangFu/projects/SAFELI/SUSHI.php.
 Cited on pages 103, 105.
- [37] Vijay Ganesh and David L. Dill.
 “A Decision Procedure for Bit-Vectors and Arrays”. In: *Computer Aided Verification (CAV 2007)*. Proceedings of the 19th International Conference. (Berlin, Germany, July 3–7, 2007). Edited by Werner Damm and Holger Hermanns. LNCS 4590. Springer, 2007, pages 519–531.
 Cited on page 104.
- [38] Vijay Ganesh, Adam Kiezun, Shay Artzi, Philip Guo, Pieter Hooimeijer, and Michael Ernst.
 “HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection”.
 In: *Computer Aided Verification (CAV 2011)*. Proceedings of the 23rd International Conference. (Snowbird, UT, USA, July 14–20, 2011).
 Edited by Ganesh Gopalakrishnan and Shaz Qadeer. LNCS 6806. Springer, 2011, pages 1–19.
 Cited on page 104.
- [39] Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte.
 “Unbounded Model-Checking with Interpolation for Regular Language Constraints”.
 In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*. Proceedings of the 19th International Conference. (Rome, Italy, Mar. 16–24, 2013).
 Edited by Nir Piterman and Scott A. Smolka. LNCS 7795. Springer, 2013, pages 277–291.
 Cited on pages 95, 103, 105.
- [40] Ian P. Gent, Christopher Jefferson, and Ian Miguel.
 “Watched Literals for Constraint Propagation in Minion”.
 In: *Principles and Practice of Constraint Programming (CP 2006)*. Proceedings of the 12th International Conference. (Nantes, France, Sept. 25–29, 2006).
 Edited by Frédéric Benhamou. LNCS 4204. Springer, 2006, pages 182–197.
 Cited on page 91.

- [41] Carmen Gervet.
“Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language”. In: *Constraints* 1.3 (1997), pages 191–244.
Cited on pages 76, 77, 95.
- [42] Carmen Gervet.
“Constraints over Structured Domains”. In: *Handbook of Constraint Programming*. Edited by Francesca Rossi, Peter van Beek, and Toby Walsh. Amsterdam: Elsevier, 2006. Chapter 17, pages 605–638.
Cited on pages 54, 89, 95, 106.
- [43] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang.
“JST: an automatic test generation tool for industrial Java applications with strings”. In: *International Conference on Software Engineering (ICSE 2013)*. Proceedings of the 35th International Conference. (San Francisco, CA, USA, May 18–26, 2013). Edited by David Notkin, Betty H. C. Cheng, and Klaus Pohl. IEEE / ACM, 2013, pages 992–1001.
Cited on page 105.
- [44] Keith Golden and Wanlin Pang.
“Constraint Reasoning over Strings”. In: *Principles and Practice of Constraint Programming (CP 2003)*. Proceedings of the 9th International Conference. (Kinsale, Ireland, Sept. 29–Oct. 3, 2003). Edited by Francesca Rossi. LNCS 2833. Springer, 2003, pages 377–391.
Cited on pages 98, 103, 106.
- [45] Keith Golden and Wanlin Pang.
“A Constraint-Based Planner Applied to Data Processing Domains”. In: *Principles and Practice of Constraint Programming (CP 2004)*. Proceedings of the 10th International Conference. (Toronto, Canada, Sept. 27–Oct. 1, 2004). Edited by Mark Wallace. LNCS 3258. Springer, 2004, page 815.
Cited on page 106.
- [46] Arnaud Gotlieb, Tristan Denmat, and Nadjib Lazaar.
“Constraint-based Reachability”. In: *Infinity 12*. Edited by M.F. Atig and A. Rezine. EPTCS 107. 2013, pages 25–43.
Cited on pages 23, 43, 50, 51.
- [47] Renaud Hartert, Pierre Schaus, Stefano Vissicchio, and Olivier Bonaventure.
“Solving Segment Routing Problems with Hybrid Constraint Programming Techniques”. In: *Principles and Practice of Constraint Programming (CP 2015)*. Proceedings of the 21st International Conference. (Cork, Ireland). Edited by Gilles Pesant. LNCS 9255. Springer, 2015, pages 592–608.
Cited on pages 108, 186, 187.
- [48] Jun He, Pierre Flener, and Justin Pearson.
“Solving String Constraints: The Case for Constraint Programming”. In: *Principles and Practice of Constraint Programming (CP 2013)*. Proceedings of the 19th International Conference. (Uppsala, Sweden, Sept. 16–20, 2013). Edited by Christian Schulte. LNCS 8124. Springer, 2013.
Cited on pages 126, 166, 170.

- [49] Samid Hoda, Willem-Jan van Hoeve, and John N. Hooker.
 “A Systematic Approach to MDD-Based Constraint Programming”.
 In: *Principles and Practice of Constraint Programming (CP 2010)*. Proceedings of the 16th International Conference. (St. Andrews, Scotland, UK, Sept. 6–10, 2010). Edited by David Cohen. LNCS 6308. Springer, 2010, pages 266–280.
 Cited on page 28.
- [50] Pieter Hooimeijer, Benjamin Livshits, D. Molnar, Prateek Saxena, and Margus Veanes.
BEK: Modeling imperative string operations with symbolic transducers.
 Technical report MSR-TR-2010-154. Microsoft Research, 2010.
 Cited on page 106.
- [51] Pieter Hooimeijer and Westley Weimer.
 “A decision procedure for subset constraints over regular languages”.
 In: *Programming Language Design and Implementation (PLDI 2009)*. Proceedings of the 2009 ACM SIGPLAN Conference on. (Dublin, Ireland, June 15–21, 2009). Edited by Michael Hind and Amer Diwan. ACM, 2009, pages 188–198.
 Cited on pages 103, 104.
- [52] Pieter Hooimeijer and Westley Weimer.
 “StrSolve: solving string constraints lazily”.
 In: *Automated Software Engineering* 19.4 (2012), pages 531–559.
 Cited on pages 103, 105.
- [53] Peter Johnstone.
Stone Spaces. Cambridge Studies in Advanced Mathematics 3.
 Cambridge University Press, 1982.
 Cited on page 42.
- [54] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst.
 “HAMPI: a solver for string constraints”. In: *International Symposium on Software Testing and Analysis (ISSTA 2009)*. Proceedings. (Chicago, IL, USA, July 19–23, 2009). Edited by Gregg Rothermel and Laura K. Dillon. HAMPI is available from <http://people.csail.mit.edu/akiezun/hampi/>. ACM, 2009, pages 105–116.
 Cited on pages 103, 104.
- [55] Stephen Cole Kleene.
Introduction to Metamathematics. New York: Van Nostrand, 1952.
 Cited on page 150.
- [56] Nikolai Kosmatov.
 “A Constraint Solver for Sequences and its Applications”. In: *ACM Symposium on Applied Computing (SAC 2006)*. Proceedings of the 2006 ACM Symposium. (Dijon, France, Apr. 23–27, 2006). Edited by Hisham Haddad. ACM, 2006, pages 404–408.
 Cited on page 28.
- [57] Vitaly Lagoon and Peter J. Stuckey.
 “Set Domain Propagation Using ROBDDs”. In: *Principles and Practice of Constraint Programming (CP 2004)*. Proceedings of the 10th International Conference. (Toronto, Canada, Sept. 27–Oct. 1, 2004). Edited by Mark Wallace. LNCS 3258. Springer, 2004, pages 347–361.
 Cited on page 95.

- [58] Nadjib Lazaar, Nourredine Aribi, Arnaud Gotlieb, and Yahia Lebbah.
Negation for Free! Technical report RR-7749. INRIA, Oct. 2011, page 16.
 Cited on page 173.
- [59] Michel Leconte.
 “A Bounds-based Reduction Scheme for Difference Constraints”. Proceedings.
 In: *FLAIRS 1996 Workshop on Constraint-based Reasoning*. (Key West, Florida).
 1996.
 Cited on pages 52, 83, 86.
- [60] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters.
 “A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions”.
 In: *Computer Aided Verification (CAV 2014)*. Proceedings of the 26th International
 Conference. (Vienna, Austria, July 18–22, 2014).
 Edited by Armin Biere and Roderick Bloem. LNCS 8559. Springer, 2014,
 pages 646–662.
 Cited on pages 105–107, 173.
- [61] M. Lothaire.
Combinatorics on words. Cambridge Mathematical Library.
 With a foreword by Roger Lyndon and a preface by Dominique Perrin, Corrected
 reprint of the 1983 original, with a new preface by Perrin.
 Cambridge: Cambridge University Press, 1997, pages xviii+238.
 Cited on page 104.
- [62] Alan K Mackworth.
 “Consistency in Networks of Relations”.
 In: *Artificial Intelligence* 8.1 (1977), pages 99–118.
 Cited on pages 51, 66.
- [63] Michael J. Maher.
 “Propagation Completeness of Reactive Constraints”. In: *Logic Programming
 (ICLP 2002)*. Proceedings of the 18th International Conference.
 (Copenhagen, Denmark, July 29–Aug. 1, 2002). Edited by Peter J. Stuckey.
 LNCS 2401. Springer, 2002, pages 148–162.
 Cited on pages 47, 50–52.
- [64] Michael J. Maher.
 “Open Constraints in a Boundable World”.
 In: *Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2009)*.
 Proceedings of the 6th International Conference.
 (Pittsburgh, PA, USA, May 27–31, 2009).
 Edited by Willem-Jan van Hoeve and John N. Hooker. LNCS 5547. Springer, 2009,
 pages 163–177.
 Cited on pages 107, 108, 168.
- [65] G.S. Makanin.
 “The problem of solvability of equations in a free semigroup”.
 In: *Sbornik: Mathematics* 32.2 (1977), pages 129–198.
 Cited on pages 103, 104.
- [66] Kim Marriot and Peter J. Stuckey.
A MiniZinc Tutorial. 2015. URL: www.minizinc.org.
 With contributions from Leslie De Koninck and Horst Samulowitz.
 Cited on pages 15, 193.

- [67] Christopher Mears, Andreas Schutt, Peter J. Stuckey, Guido Tack, Kim Marriott, and Mark Wallace.
 “Modelling with Option Types in MiniZinc”.
 In: *Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2014)*. Proceedings of the 11th International Conference. (Cork, Ireland, May 19–23, 2014). Edited by Helmut Simonis. LNCS 8451. Springer, 2014, pages 88–103.
 Cited on page 126.
- [68] Laurent D. Michel and Pascal Van Hentenryck.
 “Constraint Satisfaction over Bit-Vectors”. In: *Principles and Practice of Constraint Programming (CP 2012)*. Proceedings of the 18th International Conference. (Québec City, QC, Canada, Oct. 8–12, 2012). Edited by Michela Milano. LNCS 7514. Springer, 2012, pages 527–543.
 Cited on pages 103, 104.
- [69] Jean-Noël Monette, Pierre Flener, and Justin Pearson.
 “Towards Solver-Independent Propagators”. In: *Principles and Practice of Constraint Programming (CP 2012)*. Proceedings of the 18th International Conference. (Québec City, QC, Canada, Oct. 8–12, 2012). Edited by Michela Milano. LNCS 7514.
 Indexical compiler software available at <http://www.it.uu.se/research/group/astra/software/indexicals>.
 Springer, 2012, pages 544–560.
 Cited on pages 24, 28, 129–131, 134, 169.
- [70] Jean-Noël Monette, Pierre Flener, and Justin Pearson.
 “A Propagator Design Framework for Constraints over Sequences”.
 In: *AAAI Conference on Artificial Intelligence (AAAI 2014)*. Proceedings of the Twenty-Eighth Conference. (Québec City, Québec, Canada, July 27–31, 2014). Edited by Carla E. Brodley and Peter Stone. AAAI Press, 2014.
 Cited on pages 28, 136.
- [71] Jean-Noël Monette, Pierre Flener, and Justin Pearson.
 “Automated Auxiliary Variable Elimination Through On-the-fly Propagator Generation”. In: *Principles and Practice of Constraint Programming (CP 2015)*. Proceedings of the 21st International Conference. (Cork, Ireland). Edited by Gilles Pesant. LNCS 9255. Springer, 2015, pages 313–329.
 Cited on pages 24, 28, 129–131.
- [72] 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)*. Proceedings of the 13th International Conference. (Providence, RI, USA, Sept. 23–27, 2007). Edited by Christian Bessière. LNCS 4741. Springer, 2007, pages 529–543.
 Cited on pages 15, 126, 170, 171.
- [73] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin.
Principles of Program Analysis. Corrected 2nd printing. Berlin: Springer, 2005.
 Cited on pages 42, 116.

- [74] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou.
 “A Constraint Solver Based on Abstract Domains”.
 In: *Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*.
 Proceedings of the 14th International Conference. (Rome, Italy, Jan. 20–22, 2013).
 Edited by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. LNCS 7737.
 Springer, 2013, pages 434–454.
 Cited on pages 50, 51, 89.
- [75] Gilles Pesant.
 “A Regular Language Membership Constraint for Finite Sequences of Variables”.
 In: *Principles and Practice of Constraint Programming (CP 2004)*. Proceedings of
 the 10th International Conference. (Toronto, Canada, Sept. 27–Oct. 1, 2004).
 Edited by Mark Wallace. LNCS 3258. Springer, 2004, pages 482–495.
 Cited on pages 103, 168.
- [76] Olivier Ponsini, Claude Michel, and Michel Rueher.
 “Refining Abstract Interpretation Based Value Analysis with Constraint Programming
 Techniques”. In: *Principles and Practice of Constraint Programming (CP 2012)*.
 Proceedings of the 18th International Conference.
 (Québec City, QC, Canada, Oct. 8–12, 2012). Edited by Michela Milano. LNCS 7514.
 Springer, 2012, pages 593–607.
 Cited on page 50.
- [77] Jean-Francois Puget.
 “Set Constraints and Cardinality Operator: Application to Symmetrical Combinatorial
 Problems”. In: *Workshop on Constraint Logic Programming (WCLP 1993)*.
 (Marseille). 1993.
 Cited on pages 76, 95.
- [78] Claude-Guy Quimper and Toby Walsh.
 “Global Grammar Constraints”. In: *Principles and Practice of Constraint
 Programming (CP 2006)*. Proceedings of the 12th International Conference.
 (Nantes, France, Sept. 25–29, 2006). Edited by Frédéric Benhamou. LNCS 4204.
 Springer, 2006, pages 751–755.
 Cited on page 103.
- [79] Claude-Guy Quimper and Toby Walsh.
 “The All-Different and Global Cardinality Constraints on Set, Multiset and Tuple
 Variables”.
 In: *International Workshop on Constraint Solving and Constraint Logic Programming
 (CSCLP 2006)*. Revised Selected and Invited Papers of the Joint ERCIM/CoLogNET
 International Workshop on Constraint Solving and Constraint Logic Programming.
 (Uppsala, Sweden, June 20–22, 2005).
 Edited by Brahim Hnich, Mats Carlsson, François Fages, and Francesca Rossi.
 LNCS 3978. Springer, 2006, pages 1–13.
 Cited on page 28.
- [80] Arcot Rajasekar.
 “Applications in Constraint Logic Programming with Strings”.
 In: *Principles and Practice of Constraint Programming (PPCP 1994)*. Second
 International Workshop, Papers. (Orcas Island, Washington, USA, May 2–4, 1994).
 Edited by Alan Borning. LNCS 874. Springer, 1994, pages 109–122.
 Cited on page 107.

- [81] Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys.
 “Symbolic execution of programs with strings”. In: *South African Institute of Computer Scientists and Information Technologists Conference (SAICSIT 2012)*. (Pretoria, South Africa, Oct. 1–3, 2012).
 Edited by Jan H. Kroeze and Ruth de Villiers. ACM, 2012, pages 139–148.
 Cited on page 105.
- [82] Jean-Charles Régin.
 “A Filtering Algorithm for Constraints of Difference in CSPs”.
 In: *AAAI Conference on Artificial Intelligence (AAAI 1994)*. Proceedings. (Seattle, WA, USA, July 31–Aug. 4, 1994). Volume 1. AAAI Press, 1994, pages 362–367.
 Cited on page 27.
- [83] Andrew Sadler and Carmen Gervet.
 “Hybrid Set Domains to Strengthen Constraint Propagation and Reduce Symmetries”.
 In: *Principles and Practice of Constraint Programming (CP 2004)*. Proceedings of the 10th International Conference. (Toronto, Canada, Sept. 27–Oct. 1, 2004).
 Edited by Mark Wallace. LNCS 3258. Springer, 2004, pages 604–618.
 Cited on page 95.
- [84] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song.
 “A Symbolic Execution Framework for JavaScript”.
 In: *Security and Privacy (S&P 2010)*. Proceedings of the 31st IEEE Symposium. (Berkeley/Oakland, California, USA, May 16–19, 2010).
 IEEE Computer Society, 2010, pages 513–528. Kaluza is available from <http://webblaze.cs.berkeley.edu/2010/kaluza/>.
 Cited on pages 107, 172.
- [85] Christian Schulte.
Programming Constraint Services: High-Level Programming of Standard and New Constraint Services. Volume 2302. Berlin / Heidelberg: Springer, 2002.
 Cited on page 144.
- [86] Christian Schulte and Mats Carlsson.
 “Finite Domain Constraint Programming Systems”.
 In: *Handbook of Constraint Programming*.
 Edited by Francesca Rossi, Peter van Beek, and Toby Walsh.
 Amsterdam: Elsevier, 2006. Chapter 14, pages 495–526.
 Cited on pages 27, 32, 144, 159.
- [87] Christian Schulte and Peter J. Stuckey.
 “When Do Bounds and Domain Propagation Lead to the Same Search Space?”
 In: *ACM Trans. Program. Lang. Syst.* 27.3 (May 2005), pages 388–425.
 Cited on pages 47, 52, 83, 84.
- [88] Christian Schulte and Peter J. Stuckey.
 “Efficient Constraint Propagation Engines”. In: *Transactions on Programming Languages and Systems* 31.1 (Dec. 2008), 2:1–2:43.
 Cited on pages 139, 162.
- [89] Christian Schulte and Guido Tack.
 “Implementing Efficient Propagation Control”.
 In: *Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a conference workshop of CP 2010*. St Andrews, UK, Sept. 2010.
 Cited on page 165.

- [90] Meinolf Sellmann.
 “The Theory of Grammar Constraints”. In: *Principles and Practice of Constraint Programming (CP 2006)*. Proceedings of the 12th International Conference. (Nantes, France, Sept. 25–29, 2006). Edited by Frédéric Benhamou. LNCS 4204. Springer, 2006, pages 530–544.
 Cited on page 103.
- [91] Gregory Allan Sidebottom.
 “A Language for Optimizing Constraint Propagation”.
 PhD thesis. Simon Fraser University, Canada, 1993.
 Cited on page 129.
- [92] Yellamraju V. Srinivas and Richard Jülig.
 “Specware: Formal Support for Composing Software”.
 In: *Mathematics of Program Construction, (MPC 1995)*.
 (Kloster Irsee, Germany, July 17–21, 1995). Edited by Bernhard Möller. LNCS 947. SPECWARE is available at <http://specware.org/>. Springer, 1995, pages 399–422.
 Cited on pages 61, 130.
- [93] Richard M. Stallman and the GCC Development Community.
Using the GNU Compiler Collection. For GCC version 4.8.5.
 Boston, MA, USA: GNU Press, 2003.
 Cited on page 160.
- [94] Guido Tack.
 “Constraint Propagation: Models, Techniques, Implementation”.
 PhD thesis. Saarland University, Germany, 2009.
 Cited on pages 47, 48, 50–52, 86, 88, 163, 165.
- [95] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy, editors.
Ada 2005 Reference Manual: Language and Standard Libraries. International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1. LNCS 4348. New York; Berlin: Springer, 2006.
 Cited on page 20.
- [96] The Gecode Team.
Gecode: A Generic Constraint Development Environment. 2006.
 URL: www.gecode.org.
 Cited on pages 136, 144, 145, 165.
- [97] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar.
 “S3: A Symbolic String Solver for Vulnerability Detection in Web Applications”.
 In: *Computer and Communications Security (CCS 2014)*. Proceedings of the 2014 ACM SIGSAC Conference. (Scottsdale, AZ, USA, Nov. 3–7, 2014).
 Edited by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM, 2014.
 Cited on pages 103, 105, 106.
- [98] Charlotte Truchet, Marie Pelleau, and Frédéric Benhamou.
 “Abstract Domains for Constraint Programming, with the Example of Octagons”.
 In: *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010)*. Proceedings of the 12th International Symposium. (Timisoara, Romania, Sept. 23–26, 2010). Edited by Tetsuo Ida, Viorel Negru, Tudor Jebelean, Dana Petcu, Stephen M. Watt, and Daniela Zaharie. IEEE Computer Society, 2010, pages 72–79.
 Cited on pages 23, 50, 51.

- [99] John W Tukey.
Exploratory data analysis.
Reading, Massachusetts, USA: Addison-Wesley Publishing Company, 1977.
Cited on page 174.
- [100] Pascal Van Hentenryck and J.P. Carillon.
“Generality Versus Specificity: An Experience with AI and OR Techniques”.
In: *AAAI Conference on Artificial Intelligence (AAAI 1988)*. Proceedings of the 7th National Conference on. (St. Paul, MN, USA, Aug. 21–26, 1988).
Edited by Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith.
AAAI Press / The MIT Press, 1988, pages 660–664.
Cited on page 91.
- [101] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville.
Design, Implementation, and Evaluation of the Constraint Language cc(FD).
Technical report CS-93-02. Brown University, Providence, USA, Jan. 1993.
Revised version: *Journal of Logic Programming*, 37(1): 139–164 (1998). Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.
Cited on pages 28, 51, 80, 129.
- [102] Margus Veanes, Nikolaj Bjørner, and Leonardo Mendonça de Moura.
“Symbolic Automata Constraint Solving”.
In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*.
Proceedings of the 17th International Conference.
(Yogyakarta, Indonesia, Oct. 10–15, 2010).
Edited by Christian G. Fermüller and Andrei Voronkov. LNCS 6397. Springer, 2010,
pages 640–654.
Cited on page 106.
- [103] Justin Yip.
“The Length-Lex Representation For Constraint Programming Over Sets”.
PhD thesis. Brown University, USA, May 2011.
Cited on page 95.
- [104] Fang Yu, Tefvik Bultan, and Oscar H. Ibarra.
“Symbolic String Verification: Combining String Analysis and Size Analysis”.
In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*.
Proceedings of the 15th International Conference. (York, UK, Mar. 22–29, 2009).
Edited by Stefan Kowalewski and Anna Philippou. LNCS 5505. Springer, 2009,
pages 322–336.
Cited on page 105.
- [105] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang.
Effective Search-space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints. Technical report. 2015.
Cited on page 105.

- [106] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh.
“Z3-str: A Z3-based string solver for web application analysis”. In: *Foundations of Software Engineering (FSE 2013)*. ACM SIGSOFT Symposium on.
Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. (Saint Petersburg, Russian Federation, Aug. 18–26, 2013).
Edited by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013, pages 114–124.
Cited on pages 103, 105, 107, 173.

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science

Editor: The Dean of the Faculty of Science

1–11: 1970–1975

12. *Lars Thofelt*: Studies on leaf temperature recorded by direct measurement and by thermography. 1975.
13. *Monica Henricsson*: Nutritional studies on *Chara globularis* Thuill., *Chara zeylanica* Willd., and *Chara haitensis* Turpin. 1976.
14. *Göran Kloow*: Studies on Regenerated Cellulose by the Fluorescence Depolarization Technique. 1976.
15. *Carl-Magnus Backman*: A High Pressure Study of the Photolytic Decomposition of Azoethane and Propionyl Peroxide. 1976.
16. *Lennart Källströmer*: The significance of biotin and certain monosaccharides for the growth of *Aspergillus niger* on rhamnose medium at elevated temperature. 1977.
17. *Staffan Renlund*: Identification of Oxytocin and Vasopressin in the Bovine Adenohypophysis. 1978.
18. *Bengt Finnström*: Effects of pH, Ionic Strength and Light Intensity on the Flash Photolysis of L-tryptophan. 1978.
19. *Thomas C. Amu*: Diffusion in Dilute Solutions: An Experimental Study with Special Reference to the Effect of Size and Shape of Solute and Solvent Molecules. 1978.
20. *Lars Tegnér*: A Flash Photolysis Study of the Thermal Cis-Trans Isomerization of Some Aromatic Schiff Bases in Solution. 1979.
21. *Stig Tormod*: A High-Speed Stopped Flow Laser Light Scattering Apparatus and its Application in a Study of Conformational Changes in Bovine Serum Albumin. 1985.
22. *Björn Varnestig*: Coulomb Excitation of Rotational Nuclei. 1987.
23. *Frans Lettenström*: A study of nuclear effects in deep inelastic muon scattering. 1988.
24. *Göran Ericsson*: Production of Heavy Hypernuclei in Antiproton Annihilation. Study of their decay in the fission channel. 1988.
25. *Fang Peng*: The Geopotential: Modelling Techniques and Physical Implications with Case Studies in the South and East China Sea and Fennoscandia. 1989.
26. *Md. Anowar Hossain*: Seismic Refraction Studies in the Baltic Shield along the Fennolora Profile. 1989.
27. *Lars Erik Svensson*: Coulomb Excitation of Vibrational Nuclei. 1989.
28. *Bengt Carlsson*: Digital differentiating filters and model based fault detection. 1989.
29. *Alexander Edgar Kavka*: Coulomb Excitation. Analytical Methods and Experimental Results on even Selenium Nuclei. 1989.
30. *Christopher Juhlin*: Seismic Attenuation, Shear Wave Anisotropy and Some Aspects of Fracturing in the Crystalline Rock of the Siljan Ring Area, Central Sweden. 1990.

31. *Torbjörn Wigren*: Recursive Identification Based on the Nonlinear Wiener Model. 1990.
32. *Kjell Janson*: Experimental investigations of the proton and deuteron structure functions. 1991.
33. *Suzanne W. Harris*: Positive Muons in Crystalline and Amorphous Solids. 1991.
34. *Jan Blomgren*: Experimental Studies of Giant Resonances in Medium-Weight Spherical Nuclei. 1991.
35. *Jonas Lindgren*: Waveform Inversion of Seismic Reflection Data through Local Optimisation Methods. 1992.
36. *Liqi Fang*: Dynamic Light Scattering from Polymer Gels and Semidilute Solutions. 1992.
37. *Raymond Munier*: Segmentation, Fragmentation and Jostling of the Baltic Shield with Time. 1993.

Prior to January 1994, the series was called *Uppsala Dissertations from the Faculty of Science*.

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science and Technology

Editor: The Dean of the Faculty of Science

- 1–14: 1994–1997. 15–21: 1998–1999. 22–35: 2000–2001. 36–51: 2002–2003.
52. *Erik Larsson*: Identification of Stochastic Continuous-time Systems. Algorithms, Irregular Sampling and Cramér-Rao Bounds. 2004.
 53. *Per Åhgren*: On System Identification and Acoustic Echo Cancellation. 2004.
 54. *Felix Wehrmann*: On Modelling Nonlinear Variation in Discrete Appearances of Objects. 2004.
 55. *Peter S. Hammerstein*: Stochastic Resonance and Noise-Assisted Signal Transfer. On Coupling-Effects of Stochastic Resonators and Spectral Optimization of Fluctuations in Random Network Switches. 2004.
 56. *Esteban Damián Avendaño Soto*: Electrochromism in Nickel-based Oxides. Coloration Mechanisms and Optimization of Sputter-deposited Thin Films. 2004.
 57. *Jenny Öhman Persson*: The Obvious & The Essential. Interpreting Software Development & Organizational Change. 2004.
 58. *Chariklia Rouki*: Experimental Studies of the Synthesis and the Survival Probability of Transactinides. 2004.
 59. *Emad Abd-Elrady*: Nonlinear Approaches to Periodic Signal Modeling. 2005.
 60. *Marcus Nilsson*: Regular Model Checking. 2005.
 61. *Pritha Mahata*: Model Checking Parameterized Timed Systems. 2005.
 62. *Anders Berglund*: Learning computer systems in a distributed project course: The what, why, how and where. 2005.
 63. *Barbara Piechocinska*: Physics from Wholeness. Dynamical Totality as a Conceptual Foundation for Physical Theories. 2005.
 64. *Pär Samuelsson*: Control of Nitrogen Removal in Activated Sludge Processes. 2005.

65. *Mats Ekman*: Modeling and Control of Bilinear Systems. Application to the Activated Sludge Process. 2005.
66. *Milena Ivanova*: Scalable Scientific Stream Query Processing. 2005.
67. *Zoran Radovic*: Software Techniques for Distributed Shared Memory. 2005.
68. *Richard Abrahamsson*: Estimation Problems in Array Signal Processing, System Identification, and Radar Imagery. 2006.
69. *Fredrik Robelius*: Giant Oil Fields – The Highway to Oil. Giant Oil Fields and their Importance for Future Oil Production. 2007.
70. *Anna Davour*: Search for low mass WIMPs with the AMANDA neutrino telescope. 2007.
71. *Magnus Ågren*: Set Constraints for Local Search. 2007.
72. *Ahmed Rezine*: Parameterized Systems: Generalizing and Simplifying Automatic Verification. 2008.
73. *Linda Brus*: Nonlinear Identification and Control with Solar Energy Applications. 2008.
74. *Peter Nauchlér*: Estimation and Control of Resonant Systems with Stochastic Disturbances. 2008.
75. *Johan Petrini*: Querying RDF Schema Views of Relational Databases. 2008.
76. *Noomene Ben Henda*: Infinite-state Stochastic and Parameterized Systems. 2008.
77. *Samson Keleta*: Double Pion Production in $dd \rightarrow \alpha\pi\pi$ Reaction. 2008.
78. *Mei Hong*: Analysis of Some Methods for Identifying Dynamic Errors-invariables Systems. 2008.
79. *Robin Strand*: Distance Functions and Image Processing on Point-Lattices With Focus on the 3D Face-and Body-centered Cubic Grids. 2008.
80. *Ruslan Fomkin*: Optimization and Execution of Complex Scientific Queries. 2009.
81. *John Airey*: Science, Language and Literacy. Case Studies of Learning in Swedish University Physics. 2009.
82. *Arvid Pohl*: Search for Subrelativistic Particles with the AMANDA Neutrino Telescope. 2009.
83. *Anna Danielsson*: Doing Physics – Doing Gender. An Exploration of Physics Students' Identity Constitution in the Context of Laboratory Work. 2009.
84. *Karin Schöning*: Meson Production in pd Collisions. 2009.
85. *Henrik Petrén*: η Meson Production in Proton-Proton Collisions at Excess Energies of 40 and 72 MeV. 2009.
86. *Jan Henry Nyström*: Analysing Fault Tolerance for ERLANG Applications. 2009.
87. *John Håkansson*: Design and Verification of Component Based Real-Time Systems. 2009.
88. *Sophie Grape*: Studies of PWO Crystals and Simulations of the $\bar{p}p \rightarrow \bar{\Lambda}\Lambda, \bar{\Lambda}\Sigma^0$ Reactions for the PANDA Experiment. 2009.
90. *Agnes Rensfelt*: Viscoelastic Materials. Identification and Experiment Design. 2010.
91. *Erik Gudmundson*: Signal Processing for Spectroscopic Applications. 2010.
92. *Björn Halvarsson*: Interaction Analysis in Multivariable Control Systems. Applications to Bioreactors for Nitrogen Removal. 2010.
93. *Jesper Bengtson*: Formalising process calculi. 2010.
94. *Magnus Johansson*: Psi-calculi: a Framework for Mobile Process Calculi. Cook your own correct process calculus – just add data and logic. 2010.
95. *Karin Rathsman*: Modeling of Electron Cooling. Theory, Data and Applications. 2010.

96. *Liselott Dominicus van den Bussche*. Getting the Picture of University Physics. 2010.
97. *Olle Engdegård*. A Search for Dark Matter in the Sun with AMANDA and IceCube. 2011.
98. *Matthias Hudl*. Magnetic materials with tunable thermal, electrical, and dynamic properties. An experimental study of magnetocaloric, multiferroic, and spin-glass materials. 2012.
99. *Marcio Costa*. First-principles Studies of Local Structure Effects in Magnetic Materials. 2012.
100. *Patrik Adlarson*. Studies of the Decay $\eta \rightarrow \pi^+ \pi^- \pi^0$ with WASA-at-COSY. 2012.
101. *Erik Thomé*. Multi-Strange and Charmed Antihyperon-Hyperon Physics for PANDA. 2012.