Compiling and Executing
Finite Domain Constraints

Björn Carlson

Computing Science Department
Uppsala University

UPPSALA 1995
Compiling and Executing
Finite Domain Constraints
Compiling and Executing
Finite Domain Constraints

Björn Carlson
Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
e-mail: bjornc@csd.uu.se

A Dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at Computing Science Department, Uppsala University.

Computing Science Department
Uppsala University
Box 311, S-751 05 Uppsala
Sweden

Uppsala Theses in Computing Science
21
ISSN 0283-339X
ISBN 91-506-1100-3

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista
Sweden

SICS Dissertation Series
18
ISSN 1101-1335
ISRN SICS-D-18-SE
Doctoral thesis at Uppsala University 1995 from the Computing Science Department

Abstract:

Finite domain constraints are used for specifying and solving complex problems concerning resource allocation, scheduling, optimization, and verification; problems which are fundamental in industrial research and development. Our thesis develops new solver algorithms, compilation techniques, and programming methods for finite domain constraints.

We define a language FDC of finite domain constraints, which is closed under implication, conjunction, disjunction, negation and cardinality. FDC embeds a fair part of existing finite domain languages and is used as source language in the thesis. As target language we use FD, a language of indexicals, i.e. functions computing domains of variables.

Two execution schemes for FDC are developed; one scheme is based solely on using indexicals to evaluate the constraints; the other is based on deep-guard concurrent constraint programming together with support for indexicals.

In the indexical scheme, constraints in FDC are compiled into indexicals, where implication is treated as blocking, and disjunction as constructive. This is made possible by the support for conditional indexicals we have added to the original proposal for FD. Hence, the scheme requires no other support than a solver for conditional indexicals.

The concurrent constraint scheme is based on AKL(FD), a deep-guard concurrent constraint programming language, which supports indexicals, and which performs conditional reasoning through entailment checking and constraint lifting. Using AKL(FD) we specify a scheme which executes implications and disjunctions in deep guards, and arithmetical constraints by conjunctions of library constraints, thereby subsuming the indexical scheme. We also develop some programming techniques specific to AKL(FD), including disjunctive programming methods.

The implementations of FD and AKL(FD) are described in detail, which include an optimized solver for indexicals, the emulator and the compiler for indexicals, the integration of FD with AKL, the search primitives and the compiler extensions added to AKL, entailment checking in AKL(FD), and constraint lifting. Also, the implementations are evaluated and shown to be competitive.

Björn Carlson, Computing Science Department, Uppsala University
Box 311 S-751 05 Uppsala, Sweden
© Björn Carlson 1995

ISSN 0283-339X
ISBN 91-506-1100-3
To Ylva
Contents

1 Introduction .............................................. 1
  1.1 Finite Domain Constraints ............................. 1
  1.2 Finite Domain Constraint Programming ................. 2
  1.3 Glass-box Systems ..................................... 3
  1.4 Our Thesis ............................................. 4
    1.4.1 The indexical scheme .............................. 4
    1.4.2 The implementation of FD ......................... 5
    1.4.3 The deep-guard scheme ......................... 5
    1.4.4 The design and implementation of AKL(FD) .... 5
    1.4.5 Summary of contributions ....................... 6
    1.4.6 Thesis overview ................................ 7
    1.4.7 Acknowledgements ................................ 8

2 Finite Domain Problems .................................. 11
  2.1 Arithmetic problems ................................ 12
    2.1.1 10 equations .................................... 12
    2.1.2 20 equations .................................... 13
    2.1.3 SEND+MORE=MONEY ............................... 14
    2.1.4 Alpha puzzle .................................... 15
    2.1.5 Five houses ...................................... 16
    2.1.6 N-Queens ....................................... 17
    2.1.7 Sudoku ........................................... 18
    2.1.8 Magic series .................................... 19
  2.2 Packing ................................................ 20
    2.2.1 Square tiling .................................... 20
  2.3 Scheduling ............................................. 21
    2.3.1 Cars .............................................. 21
    2.3.2 Anytime scheduling .............................. 23
  2.4 Finite Domain Constraint Programming ................ 26
3 Finite Domain Constraints 28
  3.1 Constraint Programming .......................... 28
  3.2 Implication Checking ................................ 29
  3.3 FD .............................................. 29
  3.4 Consistency and Entailment of Indexicals ............ 32
  3.5 Monotonicity of Indexicals ......................... 33
  3.6 FDC: a Language of Finite Domain Constraints .......... 35
    3.6.1 Arithmetic constraints ........................ 35
    3.6.2 Conjunction .................................. 36
    3.6.3 Blocking implication .......................... 36
    3.6.4 Cardinality .................................. 36
    3.6.5 Disjunction .................................. 37
    3.6.6 Negation .................................... 38
    3.6.7 Symbolic constraints .......................... 39
    3.6.8 Cumulative constraints ......................... 39
  3.7 Disjunctive Normal Form .......................... 41
  3.8 FD versus FDC ................................... 41
    3.8.1 Necessary translations ........................ 42
    3.8.2 Sufficient translations ....................... 42
  3.9 Arc-consistency ................................. 43
    3.9.1 Hyper AC-3 ................................ 44
    3.9.2 Arc-consistency of conjunctions and disjunctions 45

4 The Indexical Scheme 47
  4.1 Compilation of Arithmetic Constraints ............... 47
    4.1.1 A necessary translation ........................ 48
  4.2 Necessary Translation of Disjunctions ............... 50
  4.3 Blocking Implication ............................ 53
  4.4 Arithmetics Revisited ........................... 54
    4.4.1 Nonlinear constraints ........................ 54
    4.4.2 A necessary translation ........................ 54
    4.4.3 Full arc-consistency of linear (dis)equalities 56
  4.5 Conclusion ..................................... 57

5 Entailment Checking 58
  5.1 Linear Arithmetic Constraints ...................... 58
  5.2 Nonlinear Arithmetic Constraints ................... 59
  5.3 Constraints in Disjunctive Normal Form ............. 60
    5.3.1 Some examples ............................... 60
  5.4 Implication ..................................... 61
  5.5 Symbolic Constraints ............................. 62
    5.5.1 element ...................................... 62
    5.5.2 atmost/atleast ................................ 62
  5.6 Conclusion ..................................... 62
## CONTENTS

### 6 Implementation of FD
- 6.1 The Basic FD Solver .......................... 63
  - 6.1.1 Queue of variables .......................... 63
  - 6.1.2 Queue of indexicals ......................... 65
- 6.2 Solver Optimizations for FD ...................... 65
- 6.3 The Optimized FD Solver ......................... 69
- 6.4 Data Structures for FD .......................... 70
  - 6.4.1 The finite domain variable .................. 70
  - 6.4.2 The indexical ............................... 71
- 6.5 The Monotonicity Algorithm ...................... 72
- 6.6 The FD Emulator ................................ 72
  - 6.6.1 The emulator kernel ......................... 73
  - 6.6.2 Optimized instructions ...................... 73
- 6.7 The FD Compiler ................................. 74
  - 6.7.1 Translation into byte code .................. 74
  - 6.7.2 Additional processing ....................... 75
  - 6.7.3 Compilation for arc-consistency .......... 76
  - 6.7.4 Compilation for entailment checking ...... 77
  - 6.7.5 Other optimizations ......................... 77

### 7 AKL
- 7.1 Introduction .................................. 79
- 7.2 AKL: the Language ................................ 80
  - 7.2.1 The language ............................... 80
  - 7.2.2 An example ................................ 82
- 7.3 Constraint stores in AKL ................. 84
  - 7.3.1 Constraint stores ....................... 85
  - 7.3.2 Constraint operations .................. 86
- 7.4 Conclusion .................................. 88

### 8 AKL(FD)
- 8.1 Introduction .................................. 89
- 8.2 The design of AKL(FD) ......................... 90
- 8.3 Constraint lifting in AKL(FD) ................. 91
- 8.4 FDC in AKL(FD) ................................ 92
- 8.5 Programming in AKL(FD) ....................... 93
  - 8.5.1 Example: arithmetics .................... 94
  - 8.5.2 Example: n-queens ....................... 94
  - 8.5.3 Example: atmost ......................... 95
  - 8.5.4 Example: element ....................... 97
  - 8.5.5 Example: and ............................ 98
- 8.6 Disjunction in AKL(FD) ....................... 99
  - 8.6.1 Speculative disjunction .................. 99
  - 8.6.2 Cardinality-based disjunction ......... 100
  - 8.6.3 Weak constructive disjunction .......... 101
8.6.4 Constraint lifting disjunction .................................. 103
8.7 Conclusion ............................................................... 103

9 Implementation of AKL(FD) .............................................. 104
  9.1 The FD Interface ....................................................... 104
     9.1.1 Indexicals ....................................................... 104
     9.1.2 Variables ........................................................ 105
     9.1.3 Constraints ....................................................... 106
  9.2 Adaption of the FD Solver ......................................... 107
     9.2.1 Locality testing ................................................ 107
     9.2.2 Installation and suspension ................................... 107
     9.2.3 Indexical propagation ......................................... 108
  9.3 Arithmetic Constraints in AKL(FD) .............................. 110
     9.3.1 Linear constraints compiled to library calls .............. 110
     9.3.2 Nonlinear constraints compiled to library calls ........ 112
  9.4 The AGENTS Compiler .............................................. 113
  9.5 Search in AGENTS .................................................... 114
     9.5.1 Copying-based search ....................................... 114
     9.5.2 Trailing-based search ....................................... 117
  9.6 Lifting domain constraints ....................................... 119
  9.7 Conclusion ............................................................... 119

10 Evaluation .............................................................. 120
  10.1 Evaluating AGENTS with Trailing .............................. 121
     10.1.1 Evaluating the optimizations ................................ 122
     10.1.2 Comparing with clp(FD) .................................... 126
  10.2 Evaluating AGENTS with Copying ............................... 129
     10.2.1 Comparing with Oz .......................................... 129
     10.2.2 Evaluating the disjunctive constraints ................. 132

11 Related Work .......................................................... 137
  11.1 Constraint Satisfaction Problems ............................. 137
      11.1.1 Scheduling problems ...................................... 138
  11.2 Constraint Programming Systems ............................. 138
      11.2.1 Forerunners ................................................ 139
      11.2.2 CCP .......................................................... 139
      11.2.3 CHIP ........................................................ 140
      11.2.4 cc(FD) .................................................... 141
      11.2.5 clp(FD) .................................................... 142
      11.2.6 Oz .......................................................... 143
      11.2.7 ILOG SOLVER ........................................... 143

12 Conclusion ............................................................. 145
  12.1 Thesis Contribution .............................................. 145
  12.2 Future Work ......................................................... 146
A AKL(FD) libraries .................................................. 156
   A.1 Arithmetic builtins ........................................... 156
   A.2 Symbolic builtins ............................................ 157

B AKL(FD) programs ............................................... 159
   B.1 10 and 20 Equations ....................................... 159
   B.2 SEND + MORE = MONEY .................................... 160
   B.3 Alpha .......................................................... 161
   B.4 Five Houses ................................................... 161
   B.5 N-queens ...................................................... 162
   B.6 Sudoku ........................................................ 163
   B.7 Magic series .................................................. 164
   B.8 Squares ....................................................... 165
   B.9 Cars ............................................................ 168
   B.10 Anytime scheduling ....................................... 170
   B.11 N-queens with Disjunctive Member ....................... 172
List of Figures

1.1 Chapter dependencies ........................................ 8
2.1 Sudoku puzzle .................................................. 18
2.2 Sudoku puzzle in solved form ................................ 19
2.3 Squares tiled .................................................... 20
2.4 Schematic view of a simple reprographic machine ........ 24
3.1 Syntax of range expressions ................................. 30
3.2 Cumulative constraint ......................................... 40
7.1 Syntax of statements in AKL .............................. 80
8.1 Syntax of AKL(FD) statements ............................. 90
9.1 A minimal set of library constraints ...................... 111
9.2 A minimal set of nonlinear library constraints .......... 113
9.3 Choice splitting ................................................ 114
10.1 Queens plot of AGENTS versus clp(FD) .................. 127
10.2 Magic series plot of AGENTS versus clp(FD) ............ 128
10.3 Reprographic schedule plot of AGENTS versus clp(FD) . 129
10.4 Scatter plot of AGENTS versus clp(FD) .................. 130
10.5 Queens plot of AGENTS versus Oz ....................... 132
10.6 Magic series plot of AGENTS versus Oz ................ 133
10.7 Squares plot of AGENTS versus Oz ..................... 134
10.8 Scatter plot of AGENTS versus Oz ..................... 135
## List of Tables

2.1 Car classes ................................................. 22  
2.2 Capacity frequencies ................................. 22  
3.1 Entailment/Inconsistency of $x$ in $r$ in a store $\sigma$ ............... 32  
3.2 Monotonicity of linear terms ........................ 33  
3.3 Monotonicity of nonlinear terms ..................... 33  
3.4 Monotonicity of ranges ............................... 34  
3.5 Negating arithmetic constraints ...................... 38  
4.1 Upper and lower bounds of linear terms ............... 47  
4.2 Translation of arithmetic constraints ................. 49  
4.3 Upper and lower bounds of nonlinear products ....... 54  
4.4 Translation of consistency constraints ................. 55  
5.1 Translation of entailment constraints ................. 59  
6.1 Pruning update ........................................... 68  
6.2 Byte code instructions ................................. 73  
6.3 Optimized byte code instructions ..................... 74  
6.4 Translation of ranges to byte code ................... 75  
6.5 Translation of terms to byte code .................... 75  
6.6 Monotonicity of symbolic ranges ...................... 76  
6.7 Peephole translation of ranges to byte code .......... 76  
10.1 Performance data for solver without optimizations .......... 122  
10.2 Performance data for solver with entailment marking .......... 122  
10.3 Performance data for solver using time stamps .......... 123  
10.4 Performance data for solver with dependency checking .......... 124  
10.5 Performance data for solver with all optimizations enabled .......... 125  
10.6 100*(optimizations on/optimizations off) ............ 125  
10.7 AGENTS versus clp(FD) (ms.) ........................... 126  
10.8 Scheduling jobs ........................................... 127  
10.9 AGENTS versus Oz (ms.) ............................... 131  
10.10 Square problems ........................................ 131
10.11 Bridge-project ........................................................................ 134
10.12 Squares 8 ............................................................................. 134
10.13 Squares 17 ........................................................................... 135
10.14 8-queens with member constraint ........................................... 135
Chapter 1

Introduction

Finite domain constraints are introduced and their use for programming is explained. We state our thesis topic, summarize our contributions, and outline the rest of the thesis.

1.1 Finite Domain Constraints

Packing, scheduling, optimization, and verification problems are common and important in industrial research and development. A large class of those problems can be described by logical propositions over integer relations, where all variables are finitely bounded, i.e. by finite domain constraints. They are trivially decidable, however, many finite domain problems are NP-complete. Thus, no general efficient method exists which solves such problems automatically. However, in practice, many problem instances belonging to this class can in fact be solved in polynomial time [GJ79, PS82, HSD92b, Sta94].

Propagation-based algorithms [Mac77] for simplifying a finite domain problem use the logical and arithmetical semantics of the relations to eliminate inconsistent values and tuples of values from the solution space of the problem. Thereby, the size of the search space decreases, sometimes by orders of magnitude, which simplifies searching.

Combining heuristic search with propagation algorithms is one of the most efficient and general approaches to solving finite domain problems. At each step in the search, propagation algorithms are applied to prune the size of the search space, and the search procedure hence terminates faster. Of course, for certain well-analyzed problems, such as the traveling salesman problem, there exist more efficient algorithms using problem-specific knowledge, local search or approximation algorithms [PS82]. However, as a general problem-solving methodology, the combination of search with propagation is hard to beat.
For example, constraint-based search, compared to local search and approximation algorithms, allows naturally constraints to be added/removed to/from the problem specification to direct the search, or to slightly change the specification. This is particularly advantageous in industrial use where the environment is dynamic.

1.2 Finite Domain Constraint Programming

Logic programming is based on resolution and unification [Kow74, Rob65]. Unification is naturally replaced with constraint solvers for other algebraic theories [JL87], resulting in constraint logic programming. In particular, by using the inherent non-determinism of logic programming, and by integrating propagation algorithms as solvers, a programming paradigm is realized by which finite domain problems can be effectively attacked.

The basic principle in finite domain constraint programming is to use arithmetical constraints as primitives, on top of which symbolic and propositional constraints are laid. Hence, by combining arithmetical reasoning with cardinality-based, relational, conditional, and disjunctive reasoning complex scheduling, packing, optimization, and satisfiability problems can be solved efficiently [DHS+88, DSH88, Hen89, HSD92b, DC93c].

Adding concurrency notions with constraint solving [SRP91, Mah87] shows further that programming with finite domain constraints exceeds the logic programming paradigm. Constraint solving can thus be used for synchronization of and communication between processes, as well as for programming propositional and arithmetic reasoning [CJH94].

Constraint programming offers several benefits to solving finite domain problems compared to imperative programming:

- The programmer is relieved of the gritty managing of search. This is handled by the underlying computation engine.

- The programmer can focus on finding good algebraic descriptions, e.g. by experimenting with different alternative formulations. This is possible since programs are written at a high level.

- If necessary, implicit logical and arithmetical relationships can be made explicit with reasonable effort. This can be crucial to the efficiency [HD91].

- Different constraint systems can be combined which again can improve the efficiency and overall behavior. In particular, propositional and arithmetic reasoning are naturally merged [Stä94].

As a consequence:
Development time of constraint programs is short. Typically, constraint programs are between 5 and 10 times shorter than their imperative counterpart.

Maintenance and refinement of constraint programs is manageable. The constraints are given in a syntax close to traditional algebra, and hence modifying and verifying the programs are simpler than what is the case for imperative programs where the mandatory encoding in low-level data structures muddle the reading of the programs.

Programs can be kept close to what they model. Complex problems are often hard to model accurately, and constraint-based models turn out to be natural for many of these problems. Furthermore, operational descriptions of problems are typically opaque, and a constraint model is easier to interpret.

The major drawback of constraint programming as it stands today is the lack of programmer control. The constraint solver is a black box and search primitives are defined as builtins. This severely hampers productive development, since shortcomings in the programming system can prevent a complex problem to be solved efficiently. However, current research on constraint programming improves the situation.

1.3 Glass-box Systems

The glass-box approach to constraint programming consists in controlling the constraint solver at a more detailed level than what is possible in a system where the solver is provided as a black box. Constraints that are builtins in a black-box language are instead defined as programs in a glass-box language [DHS+88, HS92a, DC93b, CCD94, CJH94].

The advantages of using glass-box languages are that the programmer is given more freedom of how to specify a problem, since the constraints can be tailored with respect to the problem at hand, and that the problem can be solved more efficiently since it need not be reduced to fit the constraints of the solver [DC93c, HS92a, CJH94].

FD is a glass-box language adapted to programming finite domain constraints [HSD91]. The basic computational primitive provided is the indexical, i.e. a function computing domains of a variable. Indexicals are used for stating arithmetical dependencies between variables such that the operational reading of the indexicals maintain constraint relations.

However, FD is a primitive language and does not contain definitions of many important symbolic and propositional finite domain constraints. Such complex constraints require conditional reasoning besides the arithmetical reasoning. Hence, to enable such constraints to be programmed either the programming language in which the indexicals are embedded must support...
appropriate conditional combinators, or FD has to be extended with necessary support for conditional reasoning. Even so, however, to completely capture the arithmetical and logical reasoning of a complex constraint by a program is non-trivial and error-prone.

Therefore, a fully-fledged constraint programming language should come equipped with pre-defined support for symbolic and propositional constraints to relieve the user. This is preferably done in a glass-box system using a constraint compiler. Consequently, traditional compiler optimizations can be made use of while preserving the benefits of glass-box languages.

1.4 Our Thesis

We are concerned with compiling and executing finite domain constraints, including symbolic and propositional constraints. Thereby we show that even small languages such as FD, with little or no extensions, can support high-level and powerful constraint programming while producing efficient code.

We define a language FDC of finite domain constraints, which is closed under implication, conjunction, disjunction, negation and cardinality. FDC embeds a fair part of existing finite domain languages, e.g. we show how several important symbolic constraints are defined in FDC, and is used as source language in the thesis.

As target language we use FD, a language of indexicals, i.e. functions computing domains of variables.

Two execution schemes for FDC are developed; one scheme is based solely on using indexicals to evaluate the constraints; the other is based on deep-guard concurrent constraint programming [Sar89, Sar93, JH91] together with support for indexicals.

The latter scheme is based on AKL(FD), a deep-guard concurrent constraint language developed at SICS [Jan94, CJH94], which supports indexicals, and which performs conditional reasoning through entailment checking and constraint lifting.

1.4.1 The indexical scheme

In the indexical scheme, constraints in FDC are compiled into monotone indexicals. Implication is treated as blocking, and disjunction as constructive [HSD92a]. This is made possible by the support for conditional indexicals we have added to the original proposal for FD [HSD91, CJH94].

Furthermore, cardinality is translated into a conjunction of implications, while preserving the operational semantics of cardinality. Hence, the scheme requires no other support than a solver for conditional indexicals, and is equally applicable in a constraint logic, as well as in a concurrent constraint, programming system.
1.4 Our Thesis

1.4.2 The implementation of FD

We give a thorough description of our implementation of FD which includes the FD solver with its optimizations, the algorithm for computing monotonicity, a description of what information must be kept associated with finite domain variables and indexicals in order to realize all necessary reasoning and optimizations, the emulator for indexicals, and the compiler algorithms used for compiling indexicals.

1.4.3 The deep-guard scheme

In the deep-guard scheme, constraints in FDC are executed by AKL(FD) statements, i.e. statements in a deep-guard concurrent constraint language. Implications and disjunctions are executed by entailment checking and constraint lifting clauses, and arithmetical constraints by conjunctions of library constraints defined by monotone indexicals. Furthermore, constraints used for entailment checking are compiled into antimonotone indexicals.

The monotonicity of indexicals plays a crucial role throughout the thesis. Therefore, we treat the issue in detail, e.g. we give an algorithm which checks both the consistency and entailment of indexicals by considering their monotonicity, as well as an efficient algorithm which derives decision information used for computing the monotonicity.

1.4.4 The design and implementation of AKL(FD)

AKL (Agents Kernel Language) generalizes the basic concurrent constraint programming functionality using a small set of powerful combinators [JH91, Jan94]. The paradigm is that of agents communicating over a constraint store, but the combinators make possible also other readings, depending on the context, where agents compute functions or relations, serve as user-defined constraints, or as objects in object-oriented programs. AKL is a deep-guard language, where being deep means having a hierarchy of constraint stores, where a computation need not be affected by the failure of a subordinate store.

The AGENTS implementation of AKL [JMB+94], being developed at SICS, has rational trees as its basic constraint system, which is supported efficiently at the emulator level. Other constraint systems, such as FD are integrated using generic variables and generic constraints. That is, objects which offer methods for the services required by the emulator, such as variable binding, garbage collection, and global propagation. This enables integration of arbitrary constraint systems with reasonable efficiency.

The implementation of FD and its integration in AGENTS is described in detail. This includes an efficient adaption of the solver for indexicals, the instantiation of the generic constraint interface of AGENTS with FD, the search primitives and the compiler extensions added to AKL, entailment
checking, constraint lifting, and the AKL(FD) libraries for arithmetic and symbolic finite domain constraints.

1.4.5 Summary of contributions

In short, our contributions are as follows:

- An algorithm which checks the consistency and entailment of indexicals.
- An algorithm for computing the monotonicity of indexicals.
- An algorithm for executing blocking implication by indexicals.
- An algorithm for executing constructive disjunction by indexicals such that the indexicals perform a non-trivial approximation of full constructive disjunction.
- An algorithm for executing blocking implication by deep-guard concurrent constraint clauses, using an entailment checking algorithm based on antimotone indexicals.
- An algorithm for executing constructive disjunction by deep-guard concurrent constraint clauses performing constraint lifting, which extends the indexical scheme.
- The generic constraint interface of AKL.
- Constraint lifting as added to AKL, and its integration in AGENTS.
- The extension of AKL with FD.
- The implementation of AKL(FD), which in particular addresses the issue of hierarchical constraint stores.
- Techniques for programming with disjunctions of finite domain constraints.

The thesis is based on the following articles, of which Björn Carlson has been the main author of all but the first and the last.


1.4 Our Thesis


1.4.6 Thesis overview

The thesis contains three themes (Figure 1.1); the compilation and execution of finite domain constraints, the compilation and execution of indexicals, and the embedding of finite domain constraints and indexicals in a deep-guard concurrent constraint language.

The chapters are structured as:

Ch 1: Thesis introduction and overview.

Ch 2: An introduction to finite domain constraints used for problem solving. We cover some well-known toy-problems, and also some practically motivated problems.

Ch 3: A precise definition of finite domain constraints, indexicals, constraints, and arc-consistency. Furthermore, compilation principles are defined which justify the proposed translations.

Ch 4: The indexical compilation scheme which generates indexicals for maintaining the consistency of constraints.

Ch 5: The indexical compilation scheme which generates indexicals for maintaining the entailment of constraints.
Ch 6: A detailed explanation of our implementation of a solver for FD.

Ch 7: An overview of AKL, including its constraint store and guard models.

Ch 8: The language AKL(FD), including its (disjunctive) programming paradigms and a description of constraint lifting.

Ch 9: The implementation of AKL(FD) (AGENTS), including the adaptation of the FD solver to the constraint store model of AGENTS.

Ch 10: The evaluation of the implementations of FD and AKL(FD).

Ch 11: Related work.

Ch 12: Conclusions and future work.

App A: Examples of some library predicates of AKL(FD).

App B: Some benchmark programs of AKL(FD).

See Figure 1.1 for a graphical overview of the thesis.

1.4.7 Acknowledgements

My work is the result of cooperation, and I am fortunate to have had very talented colleagues; the people at my department and the Programming
Systems group at SICS. Whenever a question arose somebody was there to help me. This cannot be overestimated. Also, being in two environments, CSD and SICS, has energized and inspired me.

Some people deserve extra shout outs:

**Carlsson Mats**, first and foremost, my co-advisor. My work had not been possible without him. It is a great benefit to have worked with a master of Mats’ rank, there are very few of them out there. I hope we can continue working together.

**Haridi Seif**, my advisor, has been great. He has kept me going, just like an advisor should do. He is highly intuitive and guides correspondingly.

**Sverker and his agents** Sverker was the one who got me started at SICS, of which I am forever thankful. Intellectually, his sharp insights have led me right many times, and they have improved my work considerably. His staff of agents (Johan, BD, Kent, Per, Thomas, Dan, Khayri) is very competent and very much fun to be around.

**Franzén** This Torkel bear of logical and linguistical sharpness is a wonderful digester of scientific writings. His comments strictly improve the text, always.

**Saraswat Vijay** is sparkling, and the time I have spent working with him hopefully reflects this.

**Millroth and his Reform Crew** We have shared workspace for many years, and their knowledge on programming is deep. Keep spreading it!

**Tärnlund** Sten-Åke is sometimes controversial, but always inspiring, and he got me introduced to symbolic programming.

**Barklund Jonas** has always been supportive, and works so hard to get the department going.

**Mildner Per** had a strong influence on me, when I came to Uppsala to study programming. He was an early symbolic programming guru, and hopefully his programming skills carried over.

**Gabrielsson** A deep friend and a profound programmer, whom I always enjoy being around.

**Norrbom** With friends like this, life becomes more interesting.

**My parents and sisters** Better parents and sisters do not exist.

**Ylva** finally, thank you for being with me on this journey. It made everything much more pleasant.
This work was in part funded by the ESPRIT project #7195 (ACCLAIM), and I thank the partners therein for many valuable seminars and discussions. In particular, the work on clp(FD) at INRIA, and on Oz at DFKI, has been very instructive. Also, I thank Uppsala University for supporting me for many years, and for being a place where learning is easy and fun.
Chapter 2

Finite Domain Problems

In this chapter we give some examples of how finite domain constraints are used for solving problems. Three categories of problems are considered: arithmetic puzzles, packing, and scheduling.

Let us first state the common thread of reasoning in using finite domain constraints for problem solving. Given a problem $P$, a set $V$ of finite domain variables is defined, where each variable is restricted by a finite set of objects in $P$. Given $V$, constraints are extracted from $P$ which accurately model the implicit and explicit requirements of solutions to $P$. Finally, the solutions are computed combining search with propagating logical consequences of the constraints. Usually, the algorithms that are used for propagating consequences prune the domains of the variables by eliminating values that are inconsistent with the constraints (Section 3.9).

The search is conducted by first selecting a variable to which a value is assigned. If the assignment is consistent with all constraints involving the variable, some or all values, which are inconsistent with the assignment, are eliminated from the domain of other variables. This procedure is iterated until either all variables have been assigned a value, or no consistent assignment can be made.

In the latter case, backtracking occurs, thus removing some previous assignments. The backtracking then redoes a previous assignment, this time selecting another value. Again, the consistency of the assignment is checked, and if consistent, the iteration is continued. The iteration and backtracking interplay terminates when a consistent solution is found, or when all possible assignments have been tried (remember that each variable is finitely bounded).

In the following we focus on describing problems in terms of finite domain constraints, leaving the issue of search aside. This is motivated by the fact that our thesis is concerned with constraint solving and the programming/coding of finite domain constraints primarily, where search and enumeration are closely related, but separate, research issues. However,
we include a complete but simple programming example at the end of this chapter, as to give a quick feel of the whole topic. For good introductions to general constraint programming see [Hen91, HSD92b, JM94].

2.1 Arithmetic problems

In this section we give examples of how arithmetic problems can be solved by finite domain constraints. Many of these problems are also described in [Hen89].

The first two examples are included since they are part of a set of well-known benchmark problems used for evaluating finite domain solvers [HSD92a, DC93c].

2.1.1 10 equations

The following is a simple system of linear equations over the natural numbers, where the variables range between 0 and 10, with exactly one solution. The problem is thus to compute this solution.

\[
\begin{align*}
98527x_1 + 34588x_2 + 5872x_3 + 39422x_5 + 65159x_7 &= 1547604 + 30704x_4 + 29649x_6, \\
98957x_2 + 83634x_3 + 69966x_4 + 62038x_5 + 37164x_6 + 85413x_7 &= 1823553 + 93989x_1, \\
90032 + 10949x_1 + 77761x_2 + 67052x_5 &= 80197x_3 + 61944x_4 + 92964x_6 + 44530x_7, \\
73947x_1 + 84391x_3 + 81310x_5 &= 1164380 + 96253x_2 + 44247x_4 + 70582x_6 + 33054x_7, \\
13057x_3 + 42253x_4 + 77527x_5 + 96552x_7 &= 1185471 + 60152x_1 + 21103x_2 + 97932x_6, \\
1394152 + 66920x_1 + 55679x_4 &= 64234x_2 + 65337x_3 + 45581x_5 + 67707x_6 + 98038x_7, \\
68550x_1 + 27886x_2 + 31716x_3 + 73597x_4 + 38835x_7 &= 279091 + 88963x_5 + 76391x_6, \\
76132x_2 + 71860x_3 + 22770x_4 + 68211x_5 + 78587x_6 &= 480923 + 48224x_1 + 82817x_7, \\
519878 + 94198x_2 + 87234x_3 + 37498x_4 &= 71583x_1 + 25728x_5 + 25495x_6 + 70023x_7, \\
361921 + 78693x_1 + 38592x_5 + 38478x_6 &= 94129x_2 + 43188x_3 + 82528x_4 + 69025x_7
\end{align*}
\]
The following is a simple system of linear equations over the natural numbers, where the variables range between 0 and 10, with exactly one solution.

The problem is thus to compute this solution.

\[
\begin{align*}
876370 + 16105x_1 + 6704x_3 + 68610x_5 &= 0, \\
62397x_2 + 43340x_4 + 95100x_5 + 58301x_7, \\
533909 + 96722x_5 &= 0, \\
51637x_1 + 67761x_2 + 95951x_3 + 3834x_4 + 59190x_5 + 15280x_7, \\
915683 + 34121x_2 + 33488x_7 &= 0, \\
1671x_1 + 10763x_3 + 80609x_4 + 42332x_5 + 93520x_6, \\
129768 + 11119x_2 + 38875x_4 + 14413x_5 + 29234x_6 &= 0, \\
71202x_1 + 73017x_3 + 72370x_7, \\
752447 + 58412x_2 &= 0, \\
8874x_1 + 73947x_3 + 17147x_4 + 62335x_5 + 16005x_6 + 8632x_7, \\
90614 + 18810x_3 + 48219x_4 + 79785x_7 &= 0, \\
85268x_1 + 54180x_2 + 6013x_5 + 78169x_6, \\
1198280 + 45086x_1 + 4578x_3 &= 0, \\
51830x_2 + 96120x_4 + 21231x_5 + 97919x_6 + 65651x_7, \\
18465 + 64919x_1 + 59624x_4 + 75542x_5 + 47935x_7 &= 0, \\
80460x_2 + 90840x_3 + 25145x_6, \\
43525x_2 + 92298x_3 + 58630x_4 + 92590x_5 &= 0, \\
1503588 + 43277x_1 + 9372x_6 + 60227x_7, \\
47385x_5 + 97715x_3 + 69028x_5 + 76212x_6 &= 0, \\
1244857 + 16835x_1 + 12640x_4 + 81102x_7, \\
31227x_2 + 93951x_3 + 73889x_4 + 81526x_5 + 68026x_7 &= 0, \\
1410723 + 60301x_1 + 72702x_6, \\
94016x_1 + 35961x_3 + 66597x_4 &= 0, \\
25334 + 82071x_2 + 30705x_5 + 44404x_6 + 38304x_7, \\
84750x_2 + 21239x_4 + 81675x_5 &= 0, \\
277271 + 67456x_1 + 51553x_3 + 99395x_4 + 4254x_7, \\
29958x_2 + 57308x_3 + 48789x_4 + 4657x_5 + 34539x_7 &= 0, \\
249912 + 85608x_1 + 78219x_3, \\
85176x_1 + 57898x_4 + 15883x_5 + 30547x_6 + 83287x_7 &= 0.
\end{align*}
\]
\[373854 + 95332x_2 + 1268x_3,\]
\[87758x_2 + 19346x_4 + 70072x_5 + 44529x_7 =\]
\[740061 + 10343x_1 + 11782x_3 + 36991x_6,\]
\[49149x_1 + 52871x_2 + 56728x_4 =\]
\[146074 + 7132x_3 + 33576x_5 + 49530x_6 + 62089x_7,\]
\[29475x_2 + 34421x_3 + 62646x_5 + 29278x_6 =\]
\[251591 + 60113x_1 + 76870x_4 + 15212x_7,\]
\[22167 + 29101x_2 + 5513x_3 + 21219x_4 =\]
\[87039x_1 + 22128x_5 + 7276x_6 + 57308x_7,\]
\[821228 + 76706x_1 + 48614x_6 + 41906x_7 =\]
\[98205x_2 + 23445x_3 + 67921x_4 + 24111x_5\]

The above equations can be stated directly as finite domain constraints, adding the domain constraints \(x_i \in \{0, \ldots, 10\}\), where \(1 \leq i \leq 7\).

### 2.1.3 SEND + MORE = MONEY

The problem amounts to computing an assignment of numbers from 0 to 9 to the letters S, E, N, D, M, O, R, and Y, such that adding the words “SEND” and “MORE” equals “MONEY”. Furthermore, S and M cannot be assigned 0, and no two different letters can be assigned the same number. The solution to this problem, using finite domain constraints, comes out as follows.

Let \(x_s, x_e, x_n, x_d, x_m, x_o, x_r\), and \(x_y\) be variables such that each one of them is contained in the set \(\{0, \ldots, 9\}\). The constraints modeling the problem are thus:

\[x_s \neq 0\]
\[x_m \neq 0\]
\[\text{all different}([x_s, x_e, x_n, x_d, x_m, x_o, x_r, x_y, x_d])\]
\[1000x_s + 100x_e + 10x_n + x_d + 1000x_m + 100x_o + 10x_r + x_s =\]
\[10000x_m + 1000x_o + 100x_n + 10x_e + x_y\]

where all different([\(x_1, \ldots, x_d\)]) is true iff \(x_i \neq x_j\) is true, where \(i \neq j\) and 1 \(\leq i, j \leq k\).

Since, \(x_s\) and \(x_m\) must be different from 0, it follows that \(x_s, x_m \in \{1, \ldots, 9\}\).

The only solution to this problem is 9567 + 1085 = 10652, i.e. \(x_s = 9, x_e = 5, x_n = 6, x_d = 7, x_m = 1, x_o = 0, x_r = 8,\) and \(x_y = 2\).
2.1.4 Alpha puzzle

The alpha puzzle is a problem similar to the SEND+MORE=MONEY-problem, which consists of assigning values to letters such that certain equations of letter sums are true. For example, if "L" is assigned to 14 and "O" is assigned to 5, the value of "LO" is 19. The values range between 1 and 26, where no two letters can be assigned the same value. The assignments must obey the following:

<table>
<thead>
<tr>
<th>BALLET</th>
<th>45</th>
<th>POLKA</th>
<th>59</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELLO</td>
<td>43</td>
<td>QUARTET</td>
<td>50</td>
</tr>
<tr>
<td>CONCERT</td>
<td>74</td>
<td>SAXOPHONE</td>
<td>134</td>
</tr>
<tr>
<td>FLUTE</td>
<td>30</td>
<td>SCALE</td>
<td>51</td>
</tr>
<tr>
<td>FUGUE</td>
<td>50</td>
<td>SOLO</td>
<td>37</td>
</tr>
<tr>
<td>GLEE</td>
<td>66</td>
<td>SONG</td>
<td>61</td>
</tr>
<tr>
<td>JAZZ</td>
<td>58</td>
<td>SOPRANO</td>
<td>82</td>
</tr>
<tr>
<td>LYRE</td>
<td>47</td>
<td>THEME</td>
<td>72</td>
</tr>
<tr>
<td>OBOE</td>
<td>53</td>
<td>VIOLIN</td>
<td>100</td>
</tr>
<tr>
<td>OPERA</td>
<td>65</td>
<td>WALTZ</td>
<td>34</td>
</tr>
</tbody>
</table>

The problem is thus to determine the unique value of "D". Hence, each letter $l$ is associated with a variable $x_l$, where $x_l \in \{1, \ldots, 26\}$, where

$$\text{all} \text{ different}([x_A, x_B, x_C, x_D, x_E, x_F, x_G, x_H, x_I, x_J, x_K, x_L, x_M, x_N, x_O, x_P, x_Q, x_R, x_S, x_T, x_U, x_V, x_W, x_X, x_Y, x_Z])$$

must be true. Furthermore, the following equations must be satisfied.

$$
x_B + x_A + x_L + x_L + x_E + x_T = 45, \\
x_C + x_E + x_L + x_L + x_O = 43, \\
x_C + x_O + x_N + x_C + x_E + x_R + x_T = 74, \\
x_F + x_L + x_U + x_T + x_E = 30, \\
x_F + x_U + x_G + x_U + x_E = 50, \\
x_O + x_L + x_E + x_E = 66, \\
x_J + x_A + x_Z + x_Z = 58, \\
x_L + x_Y + x_R + x_E = 47, \\
x_O + x_R + x_O + x_E = 53, \\
x_O + x_P + x_E + x_R + x_A = 65, \\
x_P + x_O + x_L + x_K + x_A = 59, \\
x_Q + x_U + x_A + x_R + x_T + x_E + x_T = 50, \\
x_S + x_A + x_X + x_O + x_P + x_H + x_O + x_N + x_E = 134,
$$
\[ x_S + x_C + x_A + x_L + x_E = 51, \]
\[ x_S + x_O + x_L + x_O = 37, \]
\[ x_S + x_O + x_N + x_G = 61, \]
\[ x_S + x_O + x_P + x_R + x_A + x_N + x_O = 82, \]
\[ x_T + x_H + x_E + x_M + x_E = 72, \]
\[ x_V + x_I + x_O + x_L + x_I + x_N = 100, \]
\[ x_W + x_A + x_L + x_T + x_Z = 34 \]

Solving the above constraints will determine \( x_D \) to 16.

### 2.1.5 Five houses

The five house problem is a logical puzzle which is naturally formulated as a finite domain problem.

Five men of different nationality (England, Spain, Japan, Italy, Norway) live in the first five houses on a street. They all each have a profession (painter, diplomat, violinist, doctor, sculptor), one animal (dog, zebra, fox, snail, horse), and one favorite drink (juice, water, tea, coffee, milk), all different from the others. Each of the houses is painted in a color different from all the others (green, red, yellow, blue, white).

Furthermore:

- The Englishman lives in the red house.
- The Spaniard owns the dog.
- The Japanese is the painter.
- The Italian likes tea.
- The Norwegian lives in the leftmost house.
- The owner of the green house likes coffee.
- The green house is to the right of the white one.
- The sculptor breeds snails.
- The diplomat lives in the yellow house.
- Milk is drunk in the third house.
- The Norwegian’s house is next to the blue one.
- The violinist likes juice.
- The fox is in the house next to the doctor’s house.
• The horse is in the house next to the diplomat’s.

The problem is thus to infer who owns the zebra and who drinks water.

Let \( n_i, c_i, p_i, a_i \) and \( d_i \) range between 1 and 5, where \( i \) range between 1 and 5 denoting the corresponding house. Let \( n_i \) represent the nationalities, e.g. \( n_1 \) denotes the nationality of the person in house 1, \( c_i \) represent the colors, \( p_i \) represent the professions, \( a_i \) represent the animals, and \( d_i \) the drinks.

Hence, it must hold that \( n_i \neq n_j \), where \( i \neq j \), and similarly for \( c_i, p_i, a_i \) and \( d_i \). Furthermore, the following must be satisfied:

\[
\begin{align*}
n_1 &= c_2, \\
n_2 &= a_1, \\
n_3 &= p_1, \\
n_4 &= d_3, \\
n_5 &= 1, \\
c_1 &= d_4, \\
c_5 + 1 &= c_1, \\
p_5 &= a_4, \\
p_2 &= c_3, \\
d_5 &= 3, \\
(n_5 + 1 = c_4 \lor c_4 + 1 = n_5) \\
p_3 &= d_1, \\
(a_3 + 1 = p_4 \lor p_4 + 1 = a_3), \\
(a_5 + 1 = p_2 \lor p_2 + 1 = a_5)
\end{align*}
\]

The above constraints have only one solution, thus determining the Japanese as the owner of the zebra, and the Norwegian as the one who drinks water.

### 2.1.6 N-Queens

The N-Queens problem needs very little introduction. Given a chess board of size \( n \times n \), place \( n \) queens such that no two queens threaten each other.

This is formulated by associating one variable \( x_i \) per queen, in the range between 1 and \( n \), stating that no two queens can share column, row, or diagonal.

Assume that variable \( x_i \) is assigned to row \( i \), hence we only need to worry about the columns and the diagonals. The constraints which maintain the
“no-threat” relation are thus as follows:

\[
x_i \neq x_j, \text{ if } i \neq j, \text{ (different columns)} \\
x_i \neq x_{i+1} + 1 \neq x_{i+1} - 1 \neq x_i, 1 \leq i \leq n - 1, \text{ (different diagonals)} \\
x_i \neq x_{i+2} + 2 \neq x_{i+2} - 2 \neq x_i, 1 \leq i \leq n - 2, \\
\vdots \\
x_1 \neq x_{n-1} + n - 1 \neq x_{n-1} - (n - 1) \neq x_1
\]

2.1.7 SudoKu

Consider the SudoKu problems, common in Japanese newspapers.

Given a 9x9 grid of squares, which is partially filled with numbers between 1 and 9, fill the rest of the squares such that each row and column is a permutation of the numbers 1 to 9. Furthermore, each 3x3 square contained in the large square, starting in columns (rows) 1, 4, or 7, must also be a permutation of the numbers 1 to 9 (Figure 2.1).

\[
\begin{array}{ccc}
8 & & 5 \\
1 & 2 & 3 & 6 \\
4 & 5 & 6 & 2 \\
7 & 8 & & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
9 & & \\
2 & 6 & 5 & 4 \\
4 & 3 & 2 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & & 9 \\
\end{array}
\]

Figure 2.1: SudoKu puzzle

First, we associate a square at position \((i, j)\) with a variable, \(x_{i,j}\), \(1 \leq i, j \leq 9\), where \(x_{i,j} \in \{1, 9\}\). Then we add the following constraints to restrict the unfilled squares.

\[
\text{all\_different}([x_{1,1}, \ldots, x_{1,9}]) \\
\vdots \\
\text{all\_different}([x_{9,1}, \ldots, x_{9,9}]) \\
\text{all\_different}([x_{1,1}, \ldots, x_{9,1}]) \\
\vdots \\
\text{all\_different}([x_{1,9}, \ldots, x_{9,9}]) \\
\text{all\_different}([x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{3,1}, x_{3,2}, x_{3,3}]) \\
\vdots \\
\text{all\_different}([x_{7,7}, x_{7,8}, x_{7,9}, \ldots, x_{9,7}, x_{9,8}, x_{9,9}])
\]
2.1 Arithmetic problems

\[
\begin{array}{cccccccc}
8 & 6 & 9 & 2 & 7 & 5 & 1 & 3 & 4 \\
7 & 1 & 2 & 3 & 4 & 9 & 6 & 8 & 5 \\
3 & 4 & 5 & 6 & 1 & 8 & 9 & 2 & 7 \\
4 & 7 & 8 & 5 & 6 & 2 & 3 & 9 & 1 \\
2 & 3 & 1 & 8 & 9 & 7 & 4 & 5 & 6 \\
9 & 5 & 6 & 4 & 3 & 1 & 8 & 7 & 2 \\
1 & 2 & 7 & 9 & 8 & 6 & 5 & 4 & 3 \\
6 & 9 & 4 & 7 & 5 & 3 & 2 & 1 & 8 \\
5 & 8 & 3 & 1 & 2 & 4 & 7 & 6 & 9 \\
\end{array}
\]

Figure 2.2: Sudoku puzzle in solved form

The sudoku problem in Figure 2.1 has the solution in Figure 2.2.

2.1.8 Magic series

Consider the problem of computing magic sequences, where a sequence \( s = \langle n_0, \ldots, n_k \rangle \) is magic if \( n_i \) is a natural number, \( 0 \leq i \leq k \), and the number of occurrences of \( i \) in \( s \) equals \( n_i \). For example, the sequence \( \langle 2,0,2,0 \rangle \) is magic, whereas \( \langle 2,0,2,1 \rangle \) is not.

Suppose \( s = \langle x_0, \ldots, x_k \rangle \), and \( x_i \in \{0, \ldots, k \} \). Then

\[
 x_i = \sum_{j=0}^{k} b_{ji},
\]

where \( b_{ji} \) is defined as \((x_j = i) \leftrightarrow b_{ji} = 1 \) and \((x_j \neq i) \leftrightarrow b_{ji} = 0, 0 \leq i \leq k \), defines precisely all the magic solutions to \( s \). This follows since

- \( b_{ji} \) equals 1 iff \( x_j \) equals \( i \), and 0 iff \( x_j \) is different from \( i \). Hence,

- \( \sum_{j=0}^{k} b_{ji} \) equals the number of elements in \( s \) which equals \( i \). Thus,

- \( x_i \) must be equal to \( \sum_{j=0}^{k} b_{ji} \).

Adding the redundant constraints

\[
 \sum_{i=0}^{k} x_i = k + 1
\]

and

\[
 \sum_{i=0}^{k} ix_i = k + 1
\]

to the above improves the behavior of the constraint solver by making intrinsic logical information explicit [HD91].
2.2 Packing

Packing problems consist of placing objects in a confined area such that certain objects do not overlap, the area is not too crowded, the distance between some objects is not too large nor too short, and/or such that the accumulated weight of overlapping objects does not violate some capacity. Also, optimization conditions may be added such that the amount of empty space, for example, is minimized.

Such problems occur in loading cargo ships, placing circuits on boards, and in storing goods in terminals.

We consider a problem of placing squares inside a larger square, which serves as an archetypal problem in this category.

2.2.1 Square tiling

The problem amounts to placing a number of squares in a larger square, in such a way that the squares do not overlap and leave no empty space [HSD92a, Hen92] (Figure 2.3). Furthermore, an additional requirement is sometimes added, which forces the small squares to be of different sizes.

![Squares tiled](image)

Figure 2.3: Squares tiled

We solve the problem as follows. Each (small) square \(i\) is associated with two variables, \(x_i\) and \(y_i\), representing the bottom-left corner of the square. It follows that \(x_i, y_i \in \{1, s - s_i + 1\}\), where \(s\) is the size of the large square, and \(s_i\) is the size of square \(i\).

For each pair of squares \(i\) and \(j\), the non-overlapping constraint is phrased as

\[
x_i + s_i \leq x_j \lor x_j + s_j \leq x_i \lor y_j + s_i \leq y_j \lor y_j + s_j \leq y_i,
\]
stating that either square $i$ is on the left, on the right, below or above square $j$.

Again, there is a potential for improving search by adding redundant constraints concerning the capacity of the large square. Namely, the accumulated size of all squares covering a certain point in the large square must equal $s$ (the size of the large square), since no empty space is allowed.

This can be stated by for each point $p$ between 1 and $s$, and each square $i$, associating a boolean variable $b_i$ such that $b_i = 1$ iff square $i$ covers $p$ and $b_i = 0$ iff square $i$ does not cover $p$. That is,

\[ b_i = 1 \iff x_i \leq p \leq x_i + s_i - 1 \]

and

\[ b_i = 0 \iff x_i > p \lor p > x_i + s_i - 1. \]

Furthermore, it is important to select a good ordering of variables. Most appropriately for this problem is an ordering which guarantees to place the variables in left to right order such that no empty space is allowed [Hen92].

### 2.3 Scheduling

Scheduling problems occur whenever scarce resources are to be used by some set of mutually dependent tasks. For example, they occur in constructing time-tables for schools and hospitals, in construction planning for bridges and houses, in deciding the order of machine instructions when compiling programming languages, and in arranging sequences of operations in some production line. Furthermore, an optimization condition is typically included such that the delay in the schedule is minimized.

In the following we consider two examples: the scheduling of cars into an assembly line, and the scheduling of images to print in a reprographic machine.

#### 2.3.1 Cars

We are given an assembly line with fixed capacities for options, where an option can, for example, be a radio or some part of the engine. The capacity for option $o$ states the maximal frequency of cars on the assembly line that require $o$. Suppose further that a set of cars to assemble is given, with corresponding requirements for options. The problem is thus to schedule the cars such that the capacities are obeyed, the requirements are satisfied, and the number of empty slots on the line is minimized [DSH88].

Consider the following problem instance. There are 10 cars, divided into 6 classes with respect to their option requirements, where there are 5 options (Table 2.1). For example, cars in class 1 require options 1, 3, and 4.
Finite Domain Problems

<table>
<thead>
<tr>
<th>class</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>option 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>option 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>option 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>option 4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>option 5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># cars</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.1: Car classes

<table>
<thead>
<tr>
<th>option</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>1/2</td>
<td>2/3</td>
<td>1/3</td>
<td>2/5</td>
<td>1/5</td>
</tr>
</tbody>
</table>

Table 2.2: Capacity frequencies

The capacity frequencies are stated in Table 2.2, where a frequency of $n/m$ for option $i$ means that out of every $m$ adjacent positions on the line, only $n$ can require the option.

We specify the finite domain variables needed as: let $s_1, \ldots, s_{10} \in \{1, \ldots, 6\}$ denote the position of the cars, i.e. if $s_j = i$ the $j$th car in the schedule is of class $i$, $o_{i,j} \in \{0, 1\}$ denote whether option $i$ is assembled at position $j$, i.e. the variable is 1 if option $i$ is assembled at position $j$ and 0 otherwise, and let $l_i$ denote row $i$ in Table 2.1.

The problem is hence captured by the following constraints. First, we need to make sure that the correct number of each class is scheduled, which is guaranteed by the formula

$$n_i = \sum_{j=1}^{10} b_{j,i},$$

where $n_i$ is the number of cars in class $i$, and $b_{j,i}$ is defined as $(s_j = i) \leftrightarrow b_{j,i} = 1$ and $(s_j \neq i) \leftrightarrow b_{j,i} = 0$, $1 \leq i \leq 6$. That is, for each class the correct number of cars is scheduled. For example, for class 3 the formula becomes

$$2 = \sum_{j=1}^{10} b_{j,3},$$

i.e. two out of the ten cars must belong to class 3.

Second, we must guarantee that the capacities are not violated. This is done by the formulas

$$a_{1,j} + \cdots + a_{1,j+2} \leq 1, (1 \leq j \leq 8)$$

$$\vdots$$

$$a_{5,j} + \cdots + a_{5,j+5} \leq 1, (1 \leq j \leq 5)$$
i.e. the capacity of any option cannot be violated anywhere on the line.

Finally, the assembling of options must correspond to the scheduling of cars, i.e. if a certain position $j$ contains a car of class $i$, this must be reflected by the fact that the option which is assembled at $j$ is a member of $i$. This is stated by the formulas

\[ \bigvee_{k=1}^{6} (s_j = k \land o_{1,j} = l_{1,k}) \]
\[ \vdots \]
\[ \bigvee_{k=1}^{6} (s_j = k \land o_{6,j} = l_{6,k}) \]

where $l_{i,k}$ is the $k$th element of $l_i$ and $1 \leq j \leq 10$. That is, for each position $j$, the option assembled at $j$ belongs to the class of the car placed at $j$.

### 2.3.2 Anytime scheduling

Reprographic machines (photo-copiers, printers, fax machines etc.) are an example of multi-pass assembly line machines. In such machines, parts are moved along the assembly line and put together until a desired output is produced. In order to save on assembly stations, the assembly line may have loops where parts are reoriented and transported back to the same station for the same operation several times.

For a reprographic machine, the output is a document consisting of printed sheets of paper. Each sheet in turn consists of a blank sheet with images on one or both sides. In the following, we will discuss a simplified but complete form of the problem of scheduling reprographic machines. While high-end machines have more complex constraints, this simpler version already represents a realistic and non-trivial machine.

### Problem statement

The task of scheduling networked reprographic machines is defined as follows [CF94]. A scheduler is given a machine description and a finite sequence of output sheets that are to be printed on the machine. The machine description is formulated in terms of constraints on sheets. The scheduler’s task is to determine which and when the operations necessary to feed and print sheets have to be performed such that the output sequence is produced.

For this thesis, we assume that the machine has one input and one output (Fig. 2.4). In between, sheets are continuously moving, i.e. once a sheet is fed into the machine, its itinerary through the machine is fixed. Thus, the itinerary is fully determined by the time at which the sheet is fed in, and the scheduler’s task is to determine the timed input sequence for a given output sequence. Note that a sheet itinerary may contain a loop, where a second image is printed on the sheet for double-sided printing. In this case, the scheduler has to leave space in the input sequence such that looping
sheets can interleave with new sheets. Also, the sheet order in the input and output sequences may be different.

The transportation and printing of sheets is constrained in various ways by the physics of the machine. A valid schedule is a timed input sequence such that the constraints are satisfied and the required output sequence is produced. An optimal schedule is a valid schedule such that no other valid schedule produces the last sheet of the output sequence earlier than the optimal one.

**Problem representation**

We are given a machine with a print component, an inverter component, and a paper path that leads from the input to the print component, the inverter component, and then either out or back to the print component (Fig. 2.4). A single-sided sheet is printed once and moved to the output, bypassing the inverter. A double-sided sheet is printed on one side, then inverted and moved back to be printed on the other side, before it is inverted again and moved to the output. Let $k$ be the time difference between first and second printing of a double-sided sheet.

The output sequence $O = o_1, \ldots, o_n$ is an ordered sequence of $n$ single- and double-sided sheets ("s" and "d" for short) specified by the user. We assume that there are at least $n$ sheets available at the machine’s input. Then the schedule to be produced by the scheduler is the explicitly timed sequence $S^e = x_1, \ldots, x_n$, where $x_i$ denotes the input time of sheet $o_i$ and ranges over points in time. (We identify input and first print time.)

As a simplification, we can assume that all sheets have to be aligned to a period $p$ (corresponding to the sheet length), i.e. all $x_i$ are multiples of $p$. Then, without loss of generality, we can choose $p = 1$. Such a schedule can conveniently be shown in an implicitly timed representation $S^i$, where element $i$ in the sequence represents what is being printed at time $i$. The alphabet for this representation is \{s,f,d,\_\}, where s, f and d denote single, front and back pages, and \_ denotes a blank spot. For example, given $O = dds$ and $k = 5$, a correct schedule would be $S^e = (1, 2, 9)$, respectively $S^i = ff\_dd\_s$ (i.e. the pages of $o_1$ are printed at times 1 and 6, the pages
of \( o_2 \) are printed at times 2 and 7, and the page of \( o_3 \) is printed at time 9. (The implicitly timed representation corresponds to the log that a sensor observing the print component once every \( p \) milliseconds would record.)

Given \( O \), the scheduler’s task is to find an \( S^* \) such that \( x_n \) is minimal and the following constraints are satisfied.

**Order constraint:** The order of \( O \) has to be preserved by \( S^* \):

\[
\forall i = 1, \ldots, n . \ y_i < y_{i+1}
\]

where \( y_i = x_i \) if \( o_i = s \), and \( y_i = x_i + k \) if \( o_i = d \) (i.e. \( y_i \) is the time when the last page of the sheet is printed). Note that, in \( S^i \), \( f \) corresponds to an \( x_i \) and \( d \) to a \( y_i \).

**Loop constraint:** A sheet returning through the loop (\( k \) after it entered the machine) must not collide with an incoming sheet:

\[
\forall i = 1, \ldots, n . \ \forall j \left( \max(i - k, 1) \leq j < i \right) . \ o_j = d \rightarrow x_i \neq x_j + k
\]

Note that for \( O = \text{sdd} \) and \( k = 5 \), both \( S^i = \text{sff_sdd} \) and \( S^i = \text{ff_sdd} \) are valid schedules.

**Inversion constraint:** The inversion of a sheet takes \( p \) milliseconds, while bypassing the inverter takes no time. Therefore, an inverted sheet (either front or backside of a double-sided sheet) cannot directly be followed by a non-inverted sheet (or else they would jam):

\[
\forall i = 1, \ldots, n . \ \forall j \left( \max(i - k, 1) \leq j < i \right) . \ o_j = s \rightarrow x_i \neq x_i + 1 \quad \land \quad \left( o_i = s \land i > 1 \right) \rightarrow o_{i-1} = d \rightarrow x_i - d + k + 1 \neq x_i
\]

So, for \( O = \text{sdd} \) and \( k = 5 \), \( S^i = \text{sff_sdd} \) is a valid schedule, while \( S^i = \text{ffs_sdd} \) is not.

**Domain:** Given the above constraints and the requirement to produce optimal schedules, we can establish (weak) lower and upper bounds for the print time:

\[
\forall i = 1, \ldots, n . \ \max(i - k, 0) + o \leq x_i \leq 2i + 2k + o
\]

where \( o \) is the current time (called the current offset).

Another constraint not arising from the machine, but from an analysis of possible optimal schedules is the distance constraint. This constraint says that — in an optimal schedule — sheets will never be more than \( k + 2 \) apart:

\[
\forall i = 1, \ldots, n - 1 . \ x_i + k + 2 \geq x_{i+1}.
\]

While this constraint is not required to define the problem, it helps enumeration by dynamically restricting the domain.
2.4 Finite Domain Constraint Programming

We end this chapter by a more detailed description of how finite domain constraints are used in Prolog-style languages, or so called Constraint Logic Programming (CLP) languages.

Consider again the Sudoku problems (Section 2.1.7). Assume we are given the unfinished square through a predicate problem(P), where P is bound to a matrix, i.e. a list of lists, containing finite domain variables or numbers in the range \{1, \ldots, 9\}. Hence, first we constrain each row in the matrix to contain all different numbers by the predicate row_constraint, then each column (column_constraint) to contain all different numbers, and, finally, each block of 3×3 squares (block_constraint) to be all different. The solutions are enumerated by nondeterministically assigning each variable a number between 1 and 9:

\[
\text{sudoku(Problem) :-}
\text{ problem(Problem),}
\text{ row_constraint(Problem),}
\text{ column_constraint(Problem),}
\text{ block_constraint(Problem),}
\text{ enumerate(Problem).}
\]

\[
\text{row_constraint([R|Rt]) :-}
\text{ all_different(R),}
\text{ row_constraint(Rt).}
\]

\[
\text{column_constraint([C1,C2,C3,C4,C5,C6,C7,C8,C9]) :-}
\text{ column_constraint(C1,C2,C3,C4,C5,C6,C7,C8,C9).}
\]

\[
\text{column_constraint([C1|C1t],[C2|C2t],[C3|C3t],[C4|C4t],}
\text{ [C5|C5t],[C6|C6t],[C7|C7t],[C8|C8t],[C9|C9t]) :-}
\text{ all_different([C1,C2,C3,C4,C5,C6,C7,C8,C9]),}
\text{ column_constraint(C1t,C2t,C3t,C4t,C5t,C6t,C7t,C8t,C9t).}
\]

\[
\text{block_constraint([C1,C2,C3,C4,C5,C6,C7,C8,C9]) :-}
\text{ block_constraint(C1,C2,C3),}
\text{ block_constraint(C4,C5,C6),}
\text{ block_constraint(C7,C8,C9).}
\]

\[
\text{block_constraint([C1,C2,C3|C1t],[C4,C5,C6|C2t],[C7,C8,C9|C3t]) :-}
\text{ all_different([C1,C2,C3,C4,C5,C6,C7,C8,C9]),}
\text{ block_constraint(C1t,C2t,C3t).}
\]

\[
\text{enumerate([]).}
\]

\[
\text{enumerate([L|P]) :-}
\]
enum(L),
enumerate(L).

elem([]).
elem([X|L]) :-
    number(X),
    enum(L).

all\_different([]).
all\_different([X|L]) :-
    different(L, X),
    all\_different(L).

different([], _).
different([Y|R], X) :-
    X \neq Y,
    different(R, X).

number(1). .. number(9).

The program is archetypal in that first the problem objects are represented by variables, i.e. the representation is generated by problem(Problem). The calls to row\_constraint(Problem), column\_constraint(Problem), and to the goal block\_constraint(Problem), constrain the variables according to the specification. Finally, the solutions are enumerated by nondeterministically searching, which is done by the call to enumerate(Problem).
Chapter 3

Finite Domain Constraints

In this chapter we define constraint programming, FD, arithmetic constraints, blocking implication, constructive disjunction, symbolic constraints, cardinality, and arc-consistency.

3.1 Constraint Programming

Constraint programming originates with the idea of using constraint solving as a means of computation. Instead of manipulating low-level data, such as pointers and integers, mathematical variables and relations are used for describing properties of objects. By using an appropriate solver algorithm, computing the extension of a set of mathematical constraints thus corresponds to computing the solution to a given problem. In this way, the programmer is relieved of much of the burden of thinking procedurally, and can focus on formulating good algebraic descriptions of the problem at hand. The benefits include shorter development time, a higher degree of reliability and correctness, and a smaller gap between problem specification and the corresponding program.

As basic notions there are constraints and constraint stores. A constraint is an algebraic formula, typically a conjunction of arithmetic expressions (such as equalities and inequalities). An assignment which satisfies a constraint is such that the constraint is true, given its interpretation, and the values assigned to the free variables therein. A constraint store is simply a set of constraints [JL87, SRP91].

A constraint store is consistent if some assignment of values to variables can be made which satisfies all the constraints in the store. A constraint is implied by a store, if any assignment which satisfies the store, satisfies the constraint.

As much of the development of constraint programming has been done inside the logic programming community, an assumption is made which
reflects the origins of the paradigm. Computations are monotone, i.e. constraints are accumulated without any retractions being made.

This gives several advantages:

- the semantics is clean and simple [JL87],
- concurrency can naturally be added [Mah87, SRP91], and
- logical combinators can be used as programming combinators, e.g. conjunction corresponds to concurrent composition, implication to conditionals, and disjunction to nondeterminism.

A computation step in a constraint programming system amounts to either solving a consistency problem for a given set of constraints, thereby possibly extending the store with constraints implied by it, or, as in the case of concurrent constraint programming, solving an implication-checking problem for a given set of constraints.

Preventing the programmer from explicitly retracting information from the store, thereby forcing the programmer to use nondeterminism in some cases, can also be a disadvantage when efficiency is critical. Consequently, constraint programming is currently not an alternative to an imperative language, such as C, when implementing an operating system or low-level network support, say.

### 3.2 Implication Checking

Common to some important constraint programming combinators, such as blocking implication and cardinality, is that constraints are checked for whether implied by the store or not. Hence, implication checking is an important notion in constraint (logic) programming. It has previously been recognized that implication checking is crucial in giving concurrent extensions of constraint programming a simple interpretation [Mah87, SRP91]. Throughout our thesis we therefore consider implication checking on equal terms with consistency checking.

### 3.3 FD

We now present a variant of the indexical language FD [HSD91]. The intuitive understanding of indexicals is that they are functional rules for maintaining consistency of arithmetic finite domain constraints.

FD is based on domain constraints and so called indexicals [HSD91]. A domain constraint is an expression \( x \in I \), where \( I \) is a set of natural numbers. A set \( \sigma \) of domain constraints is called a store. The expression \( x_\sigma \) denotes the intersection of all \( I \) s.t. \( x \in I \) is in \( \sigma \), or \( \mathcal{N} \) if there is no such \( I \). Thus, \( \sigma \) is satisfiable if no \( x_\sigma \) is empty. A variable \( x \) is determined
in \( \sigma \) if \( x_\sigma \) is a singleton set. Let \( \sigma_1 \subseteq \sigma_2 \), for stores \( \sigma_1 \) and \( \sigma_2 \), iff for all \( x, x_{\sigma_1} \subseteq x_{\sigma_2} \). By considering the equivalence classes of \( \subseteq \), i.e. \( \sigma_1 \) and \( \sigma_2 \) are equivalent iff \( \sigma_1 \subseteq \sigma_2 \) and \( \sigma_2 \subseteq \sigma_1 \), it follows easily that \( \subseteq \) is a partial order. In the following, we ignore the difference between a store and its corresponding equivalence class.

**Proposition 3.1** The set of stores, ordered by \( \subseteq \), is a lattice.

**Proof:** Obviously, \( \subseteq \) is a partial order, where the bottom element is the store \( \bot \) such that \( x_\bot = \mathcal{N} \), and the top element is the store \( \top \) such that \( x_\top = \emptyset \), for all \( x \). Furthermore, let \( \Sigma \) be a set of stores,

\[
\bigcup(\Sigma) = \{ x \in \bigcap x_\sigma; \sigma \in \Sigma \}
\]

and

\[
\bigcap(\Sigma) = \{ x \in \bigcup x_\sigma; \sigma \in \Sigma \}
\]

Let \( \sigma \in \Sigma \). For any \( x_\sigma \), obviously \( \bigcap_{\sigma' \in \Sigma} x_{\sigma'} \subseteq x_\sigma \), and \( x_\sigma \subseteq \bigcup_{\sigma' \in \Sigma} x_{\sigma'} \). Hence, \( \bigcup(\Sigma) \) is an upper bound, and \( \bigcap(\Sigma) \) is a lower bound of \( \Sigma \). Suppose \( \sigma_0 \) is an upper bound of \( \Sigma \). For each \( \sigma \in \Sigma \), thereby, \( x_{\sigma_0} \subseteq x_\sigma \), i.e. \( x_{\sigma_0} \subseteq \bigcap_{\sigma' \in \Sigma} x_{\sigma'} \), i.e. \( \bigcap(\Sigma) \) is the least upper bound of \( \Sigma \). Similarly, it follows that \( \bigcap(\Sigma) \) is the greatest lower bound.

An *indexical* (a function) has the form \( x \in r \), where \( r \) is a *range* (generated by \( R \) in Figure 3.1). When applied to a store \( \sigma \), \( x \in r \) evaluates to a domain constraint \( x \in r_\sigma \cap \mathcal{N} \), where \( r_\sigma \) is the value of \( r \) in \( \sigma \) (see below).

The value of a range \( r \) in \( \sigma \), \( r_\sigma \), is a set of integers computed as follows. The expression \texttt{dom}(y) evaluates to \( y_\sigma \). The expression \( t_1..t_2 \) is interpreted as the set \( \{ i \in \mathbb{Z} : t_{1_\sigma} \leq i \leq t_{2_\sigma} \} \). The operators \( \lor, \land \) and \( \setminus \) denote set union, intersection, and difference respectively. The conditional range \( r \Rightarrow r' \) equals \( r_\sigma \) if \( r_\sigma \neq \emptyset \) and \( \emptyset \) otherwise [CJH94]. The conditional range plays a crucial role in defining blocking implication and constructive disjunction in terms of indexicals, allowing range functions to conditionally take part in computing domain restrictions to variables (Chapter 4.2).

The expressions \( r + r' \), \( r - r' \), \( r * x \), \( r / x \), and \( r \mod x \) denote the integer operators applied pointwise, where \( x \) is a variable or a number. In the case
of division we sometimes use \(\lfloor x/y \rfloor\) to denote ordinary integer division, and \(\lfloor x/y \rfloor\) as shorthand for \((x + y - 1)/y\).

The value of a term \(t\) in \(\sigma\), \(t\), is an integer computed as follows. A number is itself. A variable \(v\) is evaluated to \(n\), if \(\nu_\sigma = \{n\}\), i.e. the value of a variable is not well-defined in all stores. The arithmetical operators are interpreted over the integers. The expressions \(\text{min}(x)\) and \(\text{max}(x)\) evaluate to the minimum and maximum of \(x_\sigma\), respectively.

In the following, if nothing else is stated, for any range \(r\) we only consider stores \(\sigma\) such that \(r_\sigma\) is well-defined. Also, where nonambiguous, we use \(t\) instead of \(t..t\). \(f\) to denote an indexical, \(x\) to denote a variable or a number, \(r\) to denote a range, and \(t\) to denote a term. The notation \(r(y)\) denotes that \(y\) occurs in \(r\). We sometimes use the notation \(t..t\). instead of \(t..\infty\) for some large constant \(\infty\), \(t\) instead of \(0..t\), and \(-r\) instead of \(0..\backslash r\). If \(r = t_1..t_2\) or \(r = \text{dom}(y)\) for some \(y\), we say that \(r\) is atomic.

Assume (finite) sets are represented by e.g. bitvectors in the following.

**Complexity 3.1** \(\sigma\) can be computed in \(O(m^{k+1})\) time, where \(m\) is the maximum size (finite) of any atomic subrange of \(r\), and \(k\) is the number of range additions (subtractions) in \(r\).

**Proof:** The ranges \(t_1..t_2\), \(\text{dom}(x)\), \(r_1 \lor r_2\), \(r_1 \land r_2\), \(r_1 \setminus r_2\), \(r_1 \Rightarrow r_2\), \(r \ast x\), and \(r/t\) can all be evaluated in \(O(m)\) time, by mapping the integer set operators to bitvector operators, e.g. intersection can be mapped to bitwise intersection, where \(r\), \(r_1\), and \(r_2\) are atomic.

We now proceed by induction over the number of range additions (subtractions). Let \(k = 1\), thus \(r = r_1 + r_2\) (or \(r = r_1 - r_2\)), where \(r_1\) and \(r_2\) can be evaluated in \(O(m)\) time. Each element in \(r_2\) can be added to \(r_1\) in \(O(m)\) time, e.g. by mapping set addition to bitwise shift, hence, \((r_1 + r_2)_\sigma\) (or \((r_1 - r_2)_\sigma\)) is \(O(m^2)\) in time.

Now, let \(k = i + 1\), for some \(i \geq 1\). Assume \(r = r_1 + r_2\) \((r_1 - r_2)\), and that the number of range additions (subtractions) in \(r_1\) equals \(j_1\) and in \(r_2\) equals \(j_2\), where \(j_1 + j_2 = i\). Hence, by the induction hypothesis, \(r_1\sigma\) is \(O(m^{j_1+1})\) in time and \(r_2\sigma\) is \(O(m^{j_2+1})\) in time. Hence, \(r_\sigma\) is \(O(m^{j_1+2} = m^{k+1})\) in time.

\(\square\)

A range \(r\) is monotone if for every pair of stores \(\sigma_1\) and \(\sigma_2\) such that \(\sigma_1 \subseteq \sigma_2\), \(r_\sigma_2 \subseteq r_\sigma_1\). \(r\) is antimonotone if for every pair of stores \(\sigma_1\) and \(\sigma_2\) such that \(\sigma_1 \supseteq \sigma_2\), \(r_\sigma_1 \subseteq r_\sigma_2\). Furthermore, \(r\) is (anti)monotone in \(\sigma\) if the above conditions hold for any \(\sigma_1\) and \(\sigma_2\) s.t. \(\sigma \subsetneq \sigma_1, \sigma_2\).

We say that \(x\) in \(r\) is monotone (antimonotone) if \(r\) is monotone (antimonotone) (see also Section 3.5). If \(r\) is both monotone and antimonotone, we say \(r\) is constant.

Note that a range such as \(\text{max}(y)\), \(\text{min}(y)\) is antimonotone in any store \(\sigma\), however, if \(y\) is determined in \(\sigma\), the range is also monotone in \(\sigma\) (and thus constant).

The final important FD notion we need to define is fixed points of indexicals [SRP91]. A store \(\sigma\) is a fixed point of \(x\) in \(r\) if \(x_\sigma \subseteq r_\sigma\).
3.4 Consistency and Entailment of Indexicals

Let \( \sigma \) be a store, then

- \( \sigma \) entails \( x \in r \) if \( r_\sigma \) is defined, and \( x_{\sigma'} \subseteq r_{\sigma'} \), for any satisfiable \( \sigma' \) such that \( \sigma \subseteq \sigma' \). Note that since \( r \) is a function we need to consider all possible extensions of \( \sigma \) when defining entailment for indexicals.

- \( x \in r \) and \( x' \in r' \) are equivalent if \( x \in r \) is entailed iff \( x' \in r' \) is entailed.

- \( x \in r \) is consistent in \( \sigma \) if for some \( \sigma' \subseteq \sigma' \), \( \sigma' \) entails \( x \in r \).

- \( x \in r \) is inconsistent in \( \sigma \) if \( x \in r \) is not consistent in \( \sigma \).

The following is immediate from the definition of fixed points.

**Proposition 3.2** \( \sigma \) entails \( x \in r \) iff for each \( \sigma' \) such that \( \sigma \subseteq \sigma' \), \( \sigma' \) is a fixed point of \( x \in r \).

The following are also immediate from the definition of monotone and antimonotone indexicals.

**Proposition 3.3** Let \( \sigma \) be a store such that \( r_\sigma \) is defined, and \( x_{\sigma'} \subseteq r_{\sigma'} \). If \( r \) is antimonotone in \( \sigma \), then \( x \in r \) is entailed in \( \sigma \).

**Proposition 3.4** Let \( \sigma \) be a store such that \( r_\sigma \) is defined, and \( x_{\sigma'} \cap r_{\sigma'} = \emptyset \). If \( r \) is monotone in \( \sigma \), then \( x \in r \) is inconsistent in \( \sigma \).

The consistency and entailment of an indexical \( x \in r \) in a store \( \sigma \) can therefore easily be checked, if the monotonicity of \( r \) in \( \sigma \) is known (see Table 3.1).

<table>
<thead>
<tr>
<th>( x \in r ) evaluated in ( \sigma )</th>
<th>( r ) monotone in ( \sigma )</th>
<th>( r ) antimonotone in ( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{\sigma} \cap r_{\sigma} = \emptyset )</td>
<td>inconsistent</td>
<td>may become entailed</td>
</tr>
<tr>
<td>( x_{\sigma} \subseteq r_{\sigma} )</td>
<td>may become inconsistent</td>
<td>entailed</td>
</tr>
<tr>
<td>( x_{\sigma} \not\subseteq (x_{\sigma} \cap r_{\sigma}) \neq \emptyset )</td>
<td>may become inconsistent</td>
<td>may become entailed</td>
</tr>
</tbody>
</table>

Table 3.1: Entailment/Inconsistency of \( x \in r \) in a store \( \sigma \)

However, Table 3.1 is not complete. Consider \( x \in \text{min}(y) \) and \( \sigma = \{ x \in \{4,5 \}, y \in \{2,3 \} \} \), where the indexical is entailed by \( \sigma \), but not antimonotone in \( \sigma \). Thus, Table 3.1 does not classify the indexical as entailed.

In Chapter 6 we show how Table 3.1 can be used as a basis for an efficient solver for indexicals.
3.5 Monotonicity of Indexicals

In Table 3.1 it is assumed that the monotonicity of a range $r$ in a store $\sigma$ is known beforehand. We now classify inductively which variables in $r$ that must be determined in $\sigma$ to guarantee that $r$ is (anti)monotone in $\sigma$. Thereby, we get an efficient procedure for computing the monotonicity of $r$ in $\sigma$.

In Table 3.4 two sets of variables, $M_r$ and $A_r$, are defined such that if all variables in $M_r$ $(A_r)$ are determined in a store $\sigma$, $r$ is (anti)monotone in $\sigma$ and in all $\sigma'$ such that $\sigma \subseteq \sigma'$.

Intuitively, the monotonicity of a range is preserved under set arithmetical operations, union, intersection, and inverted by the complement operator. The monotonicity of the interval expression $t_1..t_2$ depends on whether the terms $t_1$ and $t_2$ are increasing or decreasing expressions. The increase/decrease property of terms is preserved under addition and multiplication, and inverted in the second argument of subtraction and division.

Let $t$ be a linear term, e.g. a sum composed of linear products (Section 3.6.1), and $\nu(x)$ be a function such that $\nu(n) = \emptyset$, where $n$ is a number, and $\nu(v) = \{v\}$, where $v$ is a variable. In Table 3.2 two sets of variables, $S_t$ and $G_t$, are defined such that if all variables in $S_t$ ($G_t$) are determined in $\sigma$, then $t_\sigma \geq t_{\sigma'}$ ($t_\sigma \leq t_{\sigma'}$) for any $\sigma'$ such that $\sigma \subseteq \sigma'$.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$S_t$</th>
<th>$G_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\nu(x)$</td>
<td>$\nu(x)$</td>
</tr>
<tr>
<td>$t_1 + t_2$</td>
<td>$S_{t_1} \cup S_{t_2}$</td>
<td>$G_{t_1} \cup G_{t_2}$</td>
</tr>
<tr>
<td>$t_1 - t_2$</td>
<td>$S_{t_1} \cup G_{t_2}$</td>
<td>$G_{t_1} \cup S_{t_2}$</td>
</tr>
<tr>
<td>$x \times t$</td>
<td>$S_t \nu(x)$</td>
<td>$G_t \nu(x)$</td>
</tr>
<tr>
<td>$(-x) \times t$</td>
<td>$G_t \nu(x)$</td>
<td>$S_t \nu(x)$</td>
</tr>
<tr>
<td>$t/x$</td>
<td>$S_t \nu(x)$</td>
<td>$G_t \nu(x)$</td>
</tr>
<tr>
<td>$t/x$</td>
<td>$S_t \nu(x)$</td>
<td>$G_t \nu(x)$</td>
</tr>
<tr>
<td>$t \mod x$</td>
<td>$S_t \nu(x)$</td>
<td>$G_t \nu(x)$</td>
</tr>
<tr>
<td>$\min(x)$</td>
<td>$\nu(x)$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\max(x)$</td>
<td>$\emptyset$</td>
<td>$\nu(x)$</td>
</tr>
</tbody>
</table>

Table 3.2: Monotonicity of linear terms

Suppose $t$ is not linear. Then Table 3.3 defines $S$ and $G$ for products and quotients, where for simplicity we assume, in the table, $t_1 \sigma \geq 0$ and $t_2 \sigma \geq 0$ for all $\sigma$ (where $t_1$ and $t_2$ are defined).

<table>
<thead>
<tr>
<th>$t$</th>
<th>$S_t$</th>
<th>$G_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 \times t_2$</td>
<td>$S_{t_1} \cup S_{t_2}$</td>
<td>$G_{t_1} \cup G_{t_2}$</td>
</tr>
<tr>
<td>$t_1/t_2$</td>
<td>$S_{t_1} \cup G_{t_2}$</td>
<td>$G_{t_1} \cup S_{t_2}$</td>
</tr>
</tbody>
</table>

Table 3.3: Monotonicity of nonlinear terms
Example 3.5:

- Let $r = 1.3$. Then $M_r = A_r = \emptyset$.
- Let $r = \text{dom}(y)$. Then $M_r = \emptyset$ and $A_r = \{y\}$.
- Let $r = \text{dom}(y) \setminus \text{dom}(z)$. Then $M_r = \{z\}$ and $A_r = \{y\}$.

Lemma 3.1 Let $\sigma$ be a store. If for each $v \in S_t$ ($v \in G_t$), $v_\sigma = \{n\}$ for some $n$, $r_\sigma \geq t_\sigma$ ($t_\sigma \leq t_\sigma'$), for any $\sigma'$ such that $\sigma \subseteq \sigma'$.

Proof: By straightforward induction over $t$. □

Proposition 3.5 Let $\sigma$ be a store. If for each $v \in M_r$ ($v \in A_r$), $v_\sigma = \{n\}$ for some $n$, $r$ is (anti)monotone in $\sigma$, and in all $\sigma'$ such that $\sigma \subseteq \sigma'$.

Proof: We only prove the case of $r$ being monotone in $\sigma$. Similar reasoning proves the case of $r$ being antimonotone in $\sigma$.

We proceed by induction on $r$. In the following we assume $\sigma \subseteq \sigma'$. Let $r = t_1 \cdot t_2$. Hence, $M_r = G_t \cup S_t$. By Lemma 3.1 it follows that $t_{1 \sigma} \leq t_{1 \sigma'}$ and $t_{2 \sigma} \geq t_{2 \sigma'}$. Thus, $(t_1 \cdot t_2)_{\sigma'} \subseteq (t_1 \cdot t_2)_\sigma$, i.e. $r$ is monotone in $\sigma$.

Let $r = \text{dom}(x)$ for some $x$. It immediately follows that $r$ is monotone in any $\sigma$.

Let $r = r_1 \cdot r_2$, where $\cdot \in \{\land, \lor, +, -\}$. By the induction hypothesis, $r_{1 \sigma} \subseteq r_{1 \sigma}$ and $r_{2 \sigma} \subseteq r_{2 \sigma}$. It follows that $(r_1 \cdot r_2)_{\sigma'} \subseteq (r_1 \cdot r_2)_\sigma$.

Let $r = r_1 \Rightarrow r_2$. We consider two cases. Assume $r_{1 \sigma} = \emptyset$. By the induction hypothesis, $r_{1 \sigma'} = \emptyset$. Hence, $(r_1 \Rightarrow r_2)_{\sigma'} \subseteq (r_1 \Rightarrow r_2)_\sigma$.

Assume instead $r_{1 \sigma} \neq \emptyset$. Thereby, $r_1 \Rightarrow r_2)_{\sigma'} = r_{2 \sigma'}$. By the induction hypothesis, $r_{2 \sigma'} \subseteq r_{2 \sigma}$. Thus, $(r_1 \Rightarrow r_2)_{\sigma'} \subseteq (r_1 \Rightarrow r_2)_\sigma$.

Let $r = r_1 \setminus r_2$. Thus, $M_r = M_r \cup A_r$. By the induction hypothesis, $r_{1 \sigma} \subseteq r_{1 \sigma}$ and $r_{2 \sigma} \subseteq r_{2 \sigma'}$. Thus, $(r_1 \setminus r_2)_{\sigma'} \subseteq (r_1 \setminus r_2)_\sigma$.

Finally, let $r = r_0 \cdot x$, where $\cdot \in \{*, /\}$. By the induction hypothesis, $r_{0 \sigma'} \subseteq r_{0 \sigma}$, and, hence, $\{n : x_{\sigma'} : n \in r_{0 \sigma}\} \subseteq \{n : x_{\sigma} : n \in r_{0 \sigma}\}$. □

<table>
<thead>
<tr>
<th>$r$</th>
<th>$M_r$</th>
<th>$A_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 \cdot t_2$</td>
<td>$G_t \cup S_t$</td>
<td>$A_t \cup A_r$</td>
</tr>
<tr>
<td>$\text{dom}(x)$</td>
<td>$\emptyset$</td>
<td>$x$</td>
</tr>
<tr>
<td>$r_1 \setminus r_2$</td>
<td>$M_r \cup A_r$</td>
<td>$A_r$</td>
</tr>
<tr>
<td>$r \cdot x$</td>
<td>$M_r \lor x$</td>
<td>$A_r \lor x$</td>
</tr>
</tbody>
</table>

Table 3.4: Monotonicity of ranges
Corollary 3.1 For any store $\sigma$, $M_r = \emptyset$ implies $r$ is monotone in $\sigma$, and $A_r = \emptyset$ implies $r$ is antimonotone in $\sigma$.

Note that a range such as $\text{dom}(y) \backslash \text{dom}(y)$ is constant in any store, but $M_r = \{y\}$ and $A_r = \{y\}$, i.e. the classification is not complete. However, this is of minor importance in practice.

Complexity 3.2 The complexity of classifying the monotonicity is the complexity of the union-procedure multiplied by the numbers of operators in $r$, i.e. basically $O(v|r|)$, where $v$ is the number of variables in $r$ and $|r|$ denotes the size of $r$.

Proof: This follows immediately by computing the sets bottom-up, maintaining a lexicographic order of variables throughout, and therefore the union can be computed in order $O(v)$.

3.6 FDC: a Language of Finite Domain Constraints

We define FDC as the language of constraints where a constraint is either an arithmetic constraint (Section 3.6.1), a symbolic constraint (Section 3.6.7), a conjunction, a disjunction (Section 3.6.5) or an implication (Section 3.6.3) thereof. As is shown in Section 3.6.4 and 3.6.6, also negation and cardinality are thereby included.

FDC thus embeds a fair part of most well-known finite domain constraints, with at least one important exception. Cumulative constraints [AB93] cannot be efficiently coded in FDC, even though such constraints can be encoded in FDC in a naïve way (Section 3.6.8).

We now define inductively the semantics of constraints in FDC. Each $k$-ary constraint $c$ in FDC is a subset $R_c$ of $N_1 \times \cdots \times N_k$. A store $\sigma$ implies $c(x_1, \ldots, x_k)$ if

$$\{\langle n_1, \ldots, n_k \rangle : n_i \in x_i^\sigma, 1 \leq i \leq k \} \subseteq R_c.$$ 

A store $\sigma$ is consistent with $c$ if for some $\sigma'$, $\sigma'$ implies $c$, where $\sigma \subseteq \sigma'$. This is analogous to what is stated in Section 3.1.

Assume, for reasons of simplicity, that each $c_i$ is a $k$-ary constraint in the following.

3.6.1 Arithmetic constraints

A finite domain variable is a variable $x$ bounded by a finite set of natural numbers $I$, i.e. the domain constraint $x \in I$ holds for $x$. A linear (finite domain) constraint $x \cdot y$ is an equation ($x = y$), an inequation ($x \in \{\leq, \geq, <, >\}$), or a disequation ($x \neq y$) over linear expressions (terms) $x$ and $y$. 


where a linear expression is of the form $n_1x_1 \pm \ldots \pm n_kx_k \pm n_0$, where $n_i$ is a constant and $x_i$ a finite domain variable ($0 \leq i \leq k$).

A nonlinear finite domain constraint $x \cdot y$ is an equation ($x = y$), an inequation ($\cdot \in \{\leq,\geq,<,>\}$), or a disequation ($x \neq y$) over nonlinear expressions $x$ and $y$, where a nonlinear expression is of the form $n_1x_1y_1 \pm \ldots \pm n_kx_ky_k \pm n_0$, where $n_i$ is a constant, $x_i$ and $y_i$ are finite domain variables ($0 \leq i \leq k$). We only consider products of two variables since products of three or more variables can be transformed into products of only two variables by replacing $xyz = t$ with $xz' = t \land yz = z'$, for some new variable $z'$.

An arithmetic constraint (linear or nonlinear) $c(x_1,\ldots,x_k)$ thus defines a $k$-ary relation $R_c$ as a subset of $N_1 \times \cdots \times N_k$.

An arithmetic constraint $c$ is in normal form if

$$c \equiv S + j \cdot T + j'$$

where $S = t_1 + \ldots + t_k$, $T = t_{k+1} + \ldots + t_{n}$, each $t_i$ is equal to $n_ix_i (n_iy_i)$, where each $x_i$ ($y_i$) is a distinct variable, each coefficient $n_i$ is a natural number, $j$ and $j'$ are numbers such that $j = 0 \land j' \geq 0 \lor j \geq 0 \land j' = 0$, and $\cdot \in \{\leq,\geq,<,>\}$.

### 3.6.2 Conjunction

Conjunction is the basic combinator of constraints, in concurrent constraint, as well as in constraint logic, programming. A conjunction $c_1(x_1,\ldots,x_k) \land c_2(x_1,\ldots,x_k)$ defines a $k$-ary relation $R = R_{c_1} \cap R_{c_2}$.

### 3.6.3 Blocking implication

Conditional reasoning is important in programming. Blocking implication achieves conditional reasoning by suspending the execution of goals until certain constraints are implied by the store, i.e. given $c \rightarrow A$, $A$ is executed only when $c$ is true [HSD92a].

Blocking implication can also be viewed as an extension of the suspension mechanism freeze of Prolog [Car87] with support for entailment checking.

The semantics of an implication $c_1(x_1,\ldots,x_k) \rightarrow c_2(x_1,\ldots,x_k)$ is the $k$-ary relation $R = R_{c_1} \cup R_{c_2}$.

### 3.6.4 Cardinality

Cardinality reasoning has been shown to subsume the other logical connectives, as well as being a powerful programming primitive [HD91]. The basic construct is $\#(l,L,u)$, where $L$ is a list of constraints, $l$ is the lower bound of how many constraints in $L$ that must be true, and $u$ is the upper bound of how many constraints in $L$ that can be true, i.e. $\#(x_1,l,x_2)$ is true in $\sigma$ if the number of true constraints in $l$ is between $\max(x_1\sigma)$ and $\min(x_2\sigma)$. 
3.6 FDC: a Language of Finite Domain Constraints

Operationally, \( \#(x_1, l, x_2) \) suspends until either the number of consistent constraints in \( l \) equals \( \min(x_{1\sigma}) \), and hence, all of these constraints are added to the store, or until the number of true constraints in \( l \) equals \( \max(x_{2\sigma}) \), and hence, the negation of each of these constraints is added to the store. Furthermore, \( \max(x_{1\sigma}) (\min(x_{2\sigma})) \) is updated whenever the number of consistent (true) constraints in \( l \) goes below (above) it.

Cardinality constraints can be considered not to contain other cardinality constraints by the transformation of

\[
\#(x_1, \#(y_1, l', y_2).l, x_2)
\]

into

\[
\#(x_1, (y_1 \leq y \leq y_2).l, x_2) \land \#(y, l', y)
\]

[HD91].

In fact, \( \#(x_1, l, x_2) \) is equivalent to the formula

\[
x_1 \leq \Sigma_i b_i \leq x_2 \land \bigwedge_i (c_i \leftrightarrow b_i = 1) \land \bigwedge_i (\neg c_i \leftrightarrow b_i = 0),
\]

where \( l = c_1 \ldots c_k \theta, 1 \leq i \leq k \), and \( \leftrightarrow \) is a conjunction of two blocking implications [CCD94]. Hence, conjunction and blocking implication replace cardinality.

3.6.5 Disjunction

Disjunctive reasoning is important in constraint programming. Traditionally, disjunctions have been used for nondeterminate programming, but, recently, uses of disjunctions in determinate programs have been discovered [HD91, HSD92a, JS93, CC95]. The basic insight is that instead of using disjunction as a means for guessing computation paths, disjunction is used for pruning fruitless computation paths. Hence, the phrase constructive disjunction, although somewhat misleading since the connection to constructive logic is not clarified.

The semantics of a disjunction \( c_1(x_1, \ldots, x_k) \lor c_2(x_1, \ldots, x_k) \) is a \( k \)-ary relation \( R = R_{c_1} \cup R_{c_2} \).

We consider three operational readings of disjunction: speculative, cardinality and constructive.

Speculative

Speculative disjunction, i.e. nondeterminate disjunction, is what is used in Prolog. Executing \( \sigma \lor \sigma \) in \( \sigma \) speculatively thus means to first execute \( c_1 \) in \( \sigma \), and if failure later occurs, execute \( c_2 \) in \( \sigma \) instead. The problem with this scheme is that choices are made prematurely and that backtracking is needed to undo the effects of choices.
Cardinality-based

Cardinality-based disjunction is disjunction defined by the expression \( c_1 \lor c_2 \equiv \#(1, [c_1, c_2], 2) \) \cite{HD91}, i.e. at least one of \( c_1 \) or \( c_2 \) must be true. Hence, given a store \( \sigma \), neither \( c_1 \) nor \( c_2 \) is executed in \( \sigma \) until the other is inconsistent in \( \sigma \). The cardinality-operator is not speculative, but achieves sufficient propagation in many cases, typically for disjunctive scheduling problems.

Constructive

Constructive disjunction was proposed as a way to treat a disjunction of constraints as a constraint itself, to thereby avoid the speculative behavior, and to utilize the inherent propagation of disjunctions \cite{HSD91, HSD92a, JS93}. We only consider propagating domain constraints from a disjunction in the following.

A disjunction \( c_1(x_1, \ldots, x_k) \lor c_2(x_1, \ldots, x_k) \) applied constructively equals \( \sigma \cup \{ x_i \in I_{1i} \cup I_{2i} \}_{i=1}^k \), where \( I_{1i} = \Pi_i(R_{ci} \cap x_{1\sigma} \times \cdots \times x_{k\sigma}) \) and \( I_{2i} = \Pi_i(R_{ci} \cap x_{1\sigma} \times \cdots \times x_{k\sigma}) \), where \( \Pi_i \) denotes the projection function for the \( i \)th argument.

3.6.6 Negation

Note that the class of arithmetic constraints is closed under negation, i.e. for each arithmetic constraint \( a \), there is an arithmetic constraint equivalent to \( \neg a \), denoted by \( a^- \), defined as follows.

<table>
<thead>
<tr>
<th>a</th>
<th>a^-</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y )</td>
<td>( x \neq y )</td>
</tr>
<tr>
<td>( x \leq y )</td>
<td>( x &gt; y )</td>
</tr>
<tr>
<td>( x \geq y )</td>
<td>( x &lt; y )</td>
</tr>
<tr>
<td>( x &lt; y )</td>
<td>( x \geq y )</td>
</tr>
<tr>
<td>( x &gt; y )</td>
<td>( x \leq y )</td>
</tr>
<tr>
<td>( x \neq y )</td>
<td>( x = y )</td>
</tr>
</tbody>
</table>

Table 3.5: Negating arithmetic constraints

Let \( a \) be an arithmetic constraint, then \( a^- \) is defined by Table 3.5. The idea is simple: \( < \) is replaced by \( \geq \), \( = \) is replaced by \( \neq \), and so on. Obviously, \( a \) is true iff \( a^- \) is false, and vice versa.

Since \( a^- \) is an arithmetic constraint by definition, negation of any propositional combination of arithmetic constraints can effectively be removed by replacing \( \neg (c \land d) \) with \( \neg c \lor \neg d \), \( \neg (c \lor d) \) with \( \neg c \land \neg d \), and \( \neg \neg a \) with \( a^- \).
3.6 FDC: a Language of Finite Domain Constraints

3.6.7 Symbolic constraints

We now define symbolic constraints which are commonly used in finite domain constraint programming.

**element**

We now consider the definition of element [DSH88], where the constraint element \((i, [x_1, \ldots, x_k], v)\) is true iff \(x_i\) equals \(v\), i.e. iff \(\sqrt[j=1]{k}(i = j \land x_j = v)\) is true, where we assume \(x_j\) is a constant, \(1 \leq j \leq k\).

Operationally, the constraint can be used for eliminating inconsistent values from the domains of \(i\) and \(v\). For any \(j\) for which \(x_j \neq v\), \(j\) cannot be assigned to \(i\), and therefore it is removed from the domain of \(i\). For any element \(x\), for which there is no \(j\) in the domain of \(i\) such that \(x = x_j\), \(x\) is removed from the domain of \(v\).

**atmost/atleast/count**

Consider the constraint atmost \((u, l, v)\) which is true iff at most \(u\) elements in \(l\) are equal to \(v\), where \(l = [x_1, \ldots, x_k]\). The constraint can be defined by the formula

\[
\sum_{i=1}^{k} (b)_i \leq u
\]

where \((b)_i\) is 1 iff \(x_i = v\) is true and 0 iff \(x_i \neq v\) is true. atleast \((u, l, v)\) is similarly defined as

\[
\sum_{i=1}^{k} (b)_i \geq u,
\]

and count \((n, l, v)\) is defined as

\[
\sum_{i=1}^{k} (b)_i = n.
\]

Operationally, these constraints are special cases of the cardinality constraint.

3.6.8 Cumulative constraints

Cumulative constraints were invented to solve complex scheduling problems [AB93]. Basically, they serve as a wrapper with a logical reading around a set of highly efficient algorithms developed for scheduling. The naming “cumulative” arises from the fact that the algorithms accumulate possible values of variables, and propagate information when limits are crossed. The variables are combined to form areas which cannot be placed arbitrarily
Finite Domain Constraints

(see Figure 3.2), thus, the constraint resembles the square tiling constraint (Section 2.2.1).

Hence, the constraint

\[ \text{cumulative}([s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_1, \ldots, r_n], l) \]

where \( s_i, d_i, \) and \( r_i \) are finite domain variables, and \( l \) a constant (or a domain variable). The implicit reading of the variables involved is that \( s_i \) denotes the starting point of task or resource \( i, \) \( d_i \) the duration of task/resource \( i, \) \( r_i \) the consumption of each task/resource \( i, \) and \( l \) denotes the upper limit of the total consumption at any point. Thus, the constraint is true when the accumulated consumption at any point in the schedule does not exceed the total amount of resources, i.e. iff for each

\[ i \in \{\min(\bigcup_{i=1}^{n}\{\min(s_{i\sigma})\}), \ldots, \max(\bigcup_{i=1}^{n}\{\max(s_{i\sigma}) + \max(d_{i\sigma})\})\} \]

\[ \sum_{j \in \{k|\max(s_{k\sigma}) \leq i \leq \min(s_{k\sigma}) + \min(d_{k\sigma}) - 1\}} r_j \leq l \]

is true in \( \sigma. \)

There is an immediate encoding of the cumulative/4 constraint in FDC, which, however, does not reach the intended operational behavior. Let \( i \) be bound as above. Then cumulative([\( s_1, \ldots, s_n \]), [\( d_1, \ldots, d_n \]), [\( r_1, \ldots, r_n \]), l) is equivalent to

\[ \sum b_j r_j \leq l \]

where \( b_j \) is defined as, \( 1 \leq j \leq n: \)

\( (b_j = 1 \leftrightarrow s_j \leq i \leq s_j + d_j - 1) \land (b_j = 0 \leftrightarrow (s_j > i \lor i > s_j + d_j - 1)) \)

or as

\( (b_j = 1 \land s_j \leq i \leq s_j + d_j - 1) \lor (b_j = 0 \land (s_j > i \lor i > s_j + d_j - 1)) \)
3.7 Disjunctive Normal Form

The normalization of constraints in FDC into a disjunctive normal form, dnf for short, is now described.

In the following let $c$ and $d$ be constraints in FDC, and let $a$ be an arithmetic finite domain constraint.

Initially, cardinality constraints $(x_1, l, x_2)$ are replaced by the formula

$$x_1 \leq \sum_i b_i \leq x_2 \land \bigwedge_i (c_i \leftrightarrow b_i = 1) \land \bigwedge_i (\neg c_i \leftrightarrow b_i = 0),$$

where $l = c_1 \ldots c_k, 0$ and $1 \leq i \leq k$.

The other logical combinators are removed by applying the following rewrite rules repeatedly until no rule applies.

- **if** Implication is removed by replacing $c \rightarrow d$ with $\neg c \lor d$.
- **iff** Equivalence is removed by replacing $c \leftrightarrow d$ with $c \rightarrow d \land d \rightarrow c$.
- **not** Negation is removed by replacing $\neg(c \land d)$ with $\neg c \lor \neg d$, $\neg(c \lor d)$ with $\neg c \land \neg d$, and $\neg a$ with $a\neg$.

Finally, conjunction is distributed as

- **left** Replace $c \land (d_1 \lor d_2)$ with $(c \land d_1) \lor (c \land d_2)$.
- **right** Replace $(c_1 \lor c_2) \land d$ with $(c_1 \land d) \lor (c_2 \land d)$.

The correctness of the rules should be obvious, perhaps with the exception of the cardinality transformation. However, consider $(x_1, l, x_2)$ which is true iff at least $x_1$ and at most $x_2$ constraints in $l$ are true, which is true iff the number of $b_i$s equal to 1 lies between $x_1$ and $x_2$, i.e. iff $x_1 \leq \sum_i b_i \leq x_2$, where $b_i = 1$ iff $c_i$ is true and $b_i = 0$ iff $c_i$ is false.

Thus, given a constraint $c$ in FDC, a constraint $d$ in dnf is generated such that $c$ and $d$ are logically equivalent and $d$ is in FDC.

3.8 FD versus FDC

In this section we clarify the relationship between indexicals in FD and constraints in FDC. In particular, we show how a constraint $c$ in FDC can be defined by a set of indexicals in FD, such that either the indexicals describe necessary conditions on the store for $c$ to be true, or sufficient conditions. For a detailed description of translation schemes see Chapter 4 and 5.
3.8.1 Necessary translations

Suppose $c$ is a constraint in FDC, and $F$ is a set of indexicals. We say that a store is a fixed point of $F$ if it is a fixed point of every indexical therein. Then $F$ is a necessary translation of $c$ if every store that implies $c$ is a fixed point of $F$.

**Example 3.8.1:** Consider the equality constraint $x = y$ and the indexicals $F = \{ x \text{ in } \text{dom}(y), y \text{ in } \text{dom}(x) \}$. Then for each $\sigma$ which implies $c$ it follows that $x_\sigma = y_\sigma$, i.e. $\sigma$ is a fixed point of $F$. The same holds for $F = \{ x \text{ in } \text{dom}(y) \}$. □

The point of a necessary translation of a constraint $c$ is that it is used for checking the consistency of $c$ (Section 4). In the following, we let $F(\sigma) = \sigma \cup \{ f(\sigma) : f \in F \}$, and $F^i(\sigma) = F(\cdots(F(\sigma)))$ for $i$ occurrences of $F$.

**Proposition 3.6** Let $c$ be a constraint in FDC, $F_c$ be a necessary translation of $c$, such that each indexical in $F_c$ is monotone, $\sigma$ be a store consistent with $c$, and $\Sigma$ be defined as $\{ \sigma' : \sigma \subseteq \sigma' \land \sigma' \text{ implies } c \}$. Then for any $i$, $F_c^i(\sigma) \subseteq \cap \Sigma$.

**Proof:** Since $\subseteq$ defines a lattice, and $\sigma$ is consistent with $c$, $\cap \Sigma$ is well-defined and non-empty. Consider any $\sigma'$ such that $\sigma \subseteq \sigma'$ and $\sigma'$ implies $c$. Since $F_c$ is necessary, $F_c^i(\sigma') = \sigma'$, and since $F_c$ contains only monotone indexicals, $F_c(\sigma) \subseteq F_c(\sigma') = \sigma'$. It follows that, for any $i$, $F_c^i(\sigma) \subseteq \sigma'$. By definition of $\cap$, $F_c^i(\sigma) \subseteq \cap \Sigma$. □

**Principle of Necessary Translations:** Hence, suppose we want to compute solutions to $c$, i.e. stores which imply $c$. For any given store $\sigma$, we can, by the above, use a necessary and monotone translation $F_c$ to, given a store $\sigma$, iterate $F_c$ as $F_c^i(\sigma)$ until a fixed point $\sigma'$ is reached, and be guaranteed that all solutions to $c$ contained in $\sigma$ are also contained in $\sigma'$.

There are many alternative translations of each constraint $c$, where different translations may reach different fixed points.

**Example 3.8.1:** Consider $x = y + 1$. Two necessary and monotone translations are

$$F_1 = \{ x \text{ in } \text{dom}(y) + 1, y \text{ in } \text{dom}(x) - 1 \}$$

and

$$F_2 = \{ x \text{ in } \min(y) + 1, \max(y) + 1, y \text{ in } \min(x) - 1, \max(x) - 1 \}$$

where the second translation approximates the first by using interval reasoning. For example, consider $\sigma = \{ x \in \{1, 3, 5\}, y \in \{0, 4\} \}$ for which $F_1(\sigma) = \{ x \in \{1, 5\}, y \in \{0, 4\} \}$ and $F_2(\sigma) = \sigma$. □

3.8.2 Sufficient translations

Suppose $c$ is a constraint in FDC, and $F$ is a set of indexicals. Then $F$ is a sufficient translation of $c$ if each fixed point of $F$ implies $c$. 

3.9 Arc-consistency

Example 3.8.2: Consider \( x \leq y \) and \( F = \{0 \text{ in } 0.\min(y) - \max(x)\} \).
For each \( \sigma \) which is a fixed point of \( F \) it follows that

\[
0 \leq (\min(\sigma) - \max(\sigma))
\]

is true. Thus, \( \max(\sigma) \leq \min(\sigma) \) is true. Hence, it follows that \( x \leq y \) is true in \( \sigma \).

The intention with sufficient translations is that they can be used for checking the implication of FDC constraints (Chapter 5). Given a sufficient translation \( F_c \) and a store \( \sigma \), one simply checks whether \( F_c(\sigma) = \sigma \). By using antimonotone indexicals, if \( F_c(\sigma) = \sigma \), the test does not have to be repeated in any \( \sigma' \) such that \( \sigma \sqsubseteq \sigma' \), by Proposition 3.3.

The entailment relation of FDC is NP-complete, since for example graph-coloring, or 3SAT, can be coded by FDC constraints [GJ79]. Thus, indexicals are used for describing stronger conditions than necessary, which can efficiently be checked by a solver for FD (Section 6.3).

3.9 Arc-consistency

The basis for existing finite domain solvers is arc-consistency [Mac77], i.e. variable domains are pruned by eliminating values inconsistent with the current set of constraints.

More precisely, given a \( k \)-ary constraint in FDC, \( c(x_1, \ldots, x_k) \), and a store \( \sigma \), the constraint is arc-consistent in \( \sigma \) if for each \( n_i \in x_{i,\sigma} \) and \( i \), there exists \( n_j \in x_{j,\sigma} \), \( 1 \leq i \neq j \leq k \), such that

\[
\langle n_1, \ldots, n_k \rangle \in R_c.
\]

An arc-consistency algorithm applied to a constraint \( c(x_1, \ldots, x_k) \) hence eliminates all values \( n_i \) from \( x_{i,\sigma} \) for which \( c \) is not arc-consistent, \( 1 \leq i \leq k \).

A partial arc-consistency algorithm only eliminates a subset of all such values [DH91].

Example 3.9: Given a constraint \( c \equiv x = y + 1 \), any value \( n \) in the domain of \( x \), for which there is no value \( m \) in the domain of \( y \) such that \( n = m + 1 \) is true, is removed, and similarly for the domain of \( y \). Consider, say, the domains \( d_x = \{2, 3, 4\} \) and \( d_y = \{1, 2\} \) for \( x \) and \( y \). It follows that \( x = 4 \) is not consistent with \( c \), \( d_x \), and \( d_y \), and consequently it can be removed.

For many constraints, the semantics of the constraints can be exploited in a nontrivial way which gives highly efficient pruning algorithms [DH91, HSD92b, HDT92, AB93, CC95].

Arc-consistency is a strictly weaker notion than true consistency, i.e. a store may be arc-consistent without being consistent.

Example 3.9: Consider the constraints

\[
x \neq y \neq z \neq x
\]
and

\[ x, y, z \in \{0, 1\} \]

i.e. the variables must all be assigned different values. But, since there are three variables and only two values, all of the constraints cannot be satisfied. However, the constraints are arc-consistent.

### 3.9.1 Hyper AC-3

Finally, we include the classic AC-3 algorithm [Mac77] which stands as a model for our arc-consistency solver for indexicals (Section 6.1). We have adapted the original formulation to constraints of arbitrary arity.

In the following, consider a set of constraints over \( n \) variables as an undirected hyper-graph \( G \), i.e. a graph where arcs possibly join more than two nodes, where a node \( i \) corresponds to a variable \( x_i \), and a hyper-edge \( (x_1, \ldots, x_k) \) corresponds to a constraint \( c(x_1, \ldots, x_k) \). Let \( G \) be represented as a set of edges. The algorithm thus, given a store \( \sigma \), iterates through a queue of edges until all constraints (edges) are arc-consistent in \( \sigma \).

**Algorithm 3.9.1:**

\begin{verbatim}
revise(i, (x_1, ..., x_k), \sigma)
{ let I = \emptyset;
  for n_i \in x_i in \sigma do
    if there exist n_1 \in x_1 in \sigma, ..., n_{i-1} \in x_{i-1} in \sigma,
    n_{i+1} \in x_{i+1} in \sigma, ..., and n_k \in x_k in \sigma,
    such that R_c(n_1, ..., n_k) then set I = I \cup \{n_i\};
    if x_i in \sigma = I then return FALSE;
    else update x_i in \sigma to I and return TRUE;
}

arc_consistency(G, \sigma)
{ let Q = arcs(G);
  while Q \neq \emptyset do {
    set Q = Q \setminus \{e\};
    let e = (x_1, ..., x_k);
    for i between 1 and k do
      if revise(i, e, \sigma) then set Q = Q \cup \{e' \in G: x_i \in e' \land e \neq e'\};
  }
}

Let \( m \) be the maximum size of any domain \( x_i \), \( n \) the number of variables, \( e \) the maximum degree of any node in \( G \), and \( k \) the maximum arity of any constraint.

**Complexity 3.3** Arc-consistency can be computed in \( O(m^{k+1}ne) \) time.
3.9 Arc-consistency

Proof: Each node \(x_i\) can be revised at most \(m - 1\) times, and each time at most \(e\) edges are added to the queue. Thus, the number of dequeueings is \(O(e + n(m - 1)e) = O(mne)\). For each dequeuing, \(k\) calls to \texttt{revise} are made. For each call \texttt{revise}(\(i_r,\sigma\)), possibly \(m^{k-1}\) tests of \(R_c\) for each \(n_i\) in \(x_{i\sigma}\) are made, i.e. each call is computed in \(O(m^k)\) time. Thereby, the total runtime is \(O(km^{k+1}ne)\), or \(O(m^{k+1}ne)\) since \(k\) can be assumed to be a small constant independent of the problem.

In the special case of binary constraints, the time-complexity is thus \(O(m^2ne)\) [MF85].

It should be noted that refinements to the AC-3 algorithm above have been added, concerned with

a: relaxing the computation of \texttt{revise}, such that it becomes linear in \(k\).
This is done using so called interval-reasoning (or partial lookahead) [Hen89, HSD92a]. Instead of revising all elements in \(x_{i\sigma}\), only the minimum and maximum values in \(x_{i\sigma}\) are considered. See Chapter 4 for an example of how interval-reasoning is used for maintaining partial arc-consistency of arithmetic constraints.

b: special treatment of common constraints, such as linear arithmetic constraints, where optimizations, based on the semantics of the arithmetic predicates, are added to avoid useless calls to \texttt{revise} [HDT92] (see Chapter 6 for similar optimization techniques).

3.9.2 Arc-consistency of conjunctions and disjunctions

We now consider arc-consistency of conjunctions and disjunctions

Proposition 3.7 If a conjunction \(c_1 \land \cdots \land c_n\) is arc-consistent in \(\sigma\), then each \(c_i\) is arc-consistent in \(\sigma\). However, the converse is not true.

Proof: The proposition follows trivially from the semantics of conjunctions, and the definition of arc-consistency. A counter-example refuting the converse is given by Example 3.9.

We now show that disjunction applied constructively as defined in Section 3.6.5 is equivalent to arc-consistency applied to disjunctions.

Proposition 3.8 Let \(c\) be in \textit{dnf}, and \(c \equiv c_1 \lor \cdots \lor c_n\). Applying \(c\) constructively in \(\sigma\) equals \(\sigma\) iff \(c\) is arc-consistent in \(\sigma\).

Proof: Let \(c \equiv c_1(x_1, \ldots, x_k) \lor \cdots \lor c_n(x_1, \ldots, x_k)\), and suppose \(c\) is arc-consistent in \(\sigma\). Hence, for each \(n_i \in x_{i\sigma}\), there exists \(n_j \in x_{j\sigma}\), \(i \neq j\), such that \(\langle n_1, \ldots, n_k\rangle \in R_c = R_{c_1} \cup \cdots \cup R_{c_n}\). Thereby, for some \(m\) between 1 and \(n\), \(\langle n_1, \ldots, n_k\rangle \in R_{c_m} \cap x_{i\sigma} \times \cdots \times x_{k\sigma}\), and, thus, \(x_{i\sigma} \subseteq \bigcup_{m=1}^{n} \Pi_i(R_{c_m} \cap x_{i\sigma} \times \cdots \times x_{k\sigma})\), for each \(i\) between 1 and \(k\). Hence, we have proved the if part.
Suppose instead that applying $c$ constructively in $\sigma$ equals $\sigma$, i.e. $x_{i,\sigma} \subseteq \bigcup_{m=1}^{n} \prod_{i}(R_{c,m} \cap x_{1,\sigma} \times \cdots \times x_{k,\sigma})$, for each $i$ between 1 and $k$. Let $n_{i} \in x_{i,\sigma}$. It follows that there exists $m$ such that $\langle n_{1}, \ldots, n_{i}, \ldots, n_{k} \rangle \in R_{c,m} \cap x_{1,\sigma} \times \cdots \times x_{k,\sigma}$, i.e. $\langle n_{1}, \ldots, n_{i}, \ldots, n_{k} \rangle \in R_{c} \cap x_{1,\sigma} \times \cdots \times x_{k,\sigma}$, which proves the only-if part.
Chapter 4

The Indexical Scheme

This chapter deals with the indexical compilation scheme, which is a scheme for compiling constraints to indexicals. We give the translation rules for removing disjunctions and implications, and we describe important optimizations used for evaluating the generated indexicals. Also, we show that the translation is a necessary such (Section 3.8). The principle of necessary translations is used (Page 42), and therefore we consider monotone indexicals throughout.

4.1 Compilation of Arithmetic Constraints

In this section, we describe the compilation of linear finite domain constraints to monotone indexicals.

Let $\inf$ ($\sup$) be a function from linear terms to values which increases (decreases) as the computation progresses. That is, $\inf(t)$ ($\sup(t)$) is the smallest (largest) value that $t$ can ever get (see Table 4.1).

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\inf(t)$</th>
<th>$\sup(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\min(x)$</td>
<td>$\max(x)$</td>
</tr>
<tr>
<td>$t_1 + t_2$</td>
<td>$\inf(t_1) + \inf(t_2)$</td>
<td>$\sup(t_1) + \sup(t_2)$</td>
</tr>
<tr>
<td>$t_1 - t_2$</td>
<td>$\inf(t_1) - \sup(t_2)$</td>
<td>$\sup(t_1) - \inf(t_2)$</td>
</tr>
<tr>
<td>$nx$</td>
<td>$n \cdot \inf(x)$</td>
<td>$n \cdot \sup(x)$</td>
</tr>
</tbody>
</table>

Table 4.1: Upper and lower bounds of linear terms

The lower (upper) bound of a linear term $E$ is thus computed by $\inf(E)$ ($\sup(E)$) (see Table 4.1).

Proposition 4.1 $S_{\sup(t)} = G_{\inf(t)} = \emptyset$. 


**Proof:** A simple induction over $t$, using the definitions of $\inf(t)$, $\sup(t)$, $S_t$ and $G_t$. \qed

### 4.1.1 A necessary translation

We give a simple-minded compilation of arithmetic constraints into indexicals, used as a basis for our compilation of disjunctions in later sections (see Section 4.2).

First, it should be noted that domain constraints $x \in \{n_1, \ldots, n_k\}$ are trivially coded in FD as $x \in n_1 \lor \cdots \lor n_k$.

The compilation of an arithmetic constraint $c$ is based on deriving necessary conditions for $c$ expressed as monotone indexicals. A constraint over $k$ variables is compiled into $k$ monotone indexicals over $k - 1$ variables, which approximate the constraint by interval arithmetic reasoning, i.e. they maintain partial arc-consistency. This is similar to the distinction made between interval and domain reasoning of constraints in $\alpha(FD)$ [HSD92a]. The scheme can be modified to provide full arc-consistency (Section 4.4.3). The method is best explained by an example.

**Example 4.1.1:** The constraint

$$2x = 3y + 5$$

is rewritten twice to equivalent equations, each expressing a single variable as a function of the others:

$$2x = 3y + 5,$$

$$3y = 2x - 5$$

These equations are approximated by the indexicals:

$$x \in [(3 \cdot \min(y) + 5)/2], (3 \cdot \max(y) + 5)/2],$$

$$y \in [(2 \cdot \min(x) - 5)/3], (2 \cdot \max(x) - 5)/3]$$

The compilation of inequations and disequations is completely analogous to that of equations. In general, the idea is to rewrite a constraint over the variables $x_1 \ldots x_k$ into the equivalent constraints

$$n_1 x_1 \cdot E_1$$

$$\vdots$$

$$n_k x_k \cdot E_k$$
where \( \cdot \) is the relation symbol and then to translate them into a conjunction of monotone indexicals

\[
\begin{align*}
x_1 &\in r_1 \\
\vdots \\
x_k &\in r_k
\end{align*}
\]

that propagate information whenever the \textbf{min} or the \textbf{max} of a variable changes, where \( r_i \) is defined from \( E_i \) and \( n_i \) (see Table 4.2).

The translation rules (see Table 4.2) are obtained as follows: a necessary condition for \( nx \leq E \) is the indexical \( x \in \ldots[\sup(E)/n] \); a necessary condition for \( nx \geq E \) is the indexical \( x \in [\inf(E)/n] \); and the following equivalences hold:

\[
\begin{align*}
x = y &\equiv x \leq y \land x \geq y \\
x < y &\equiv x \leq y - 1 \\
x > y &\equiv x \geq y + 1 \\
x \neq y &\equiv x < y \lor x > y \\
x \in r_1 \land x \in r_2 &\equiv x \in r_1 \land r_2 \\
x \in r_1 \lor x \in r_2 &\equiv x \in r_1 \lor r_2 \\
x \in \ldots(i - 1) \lor (j + 1).. &\equiv x \in \ldots-i..j
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \cdot )</th>
<th>( r_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( = )</td>
<td>( \lfloor \inf(E_i)/n_i \rfloor \ldots \lfloor \sup(E_i)/n_i \rfloor )</td>
</tr>
<tr>
<td>( \leq )</td>
<td>( \ldots \lfloor \sup(E_i)/n_i \rfloor )</td>
</tr>
<tr>
<td>( \geq )</td>
<td>( \lfloor \inf(E_i)/n_i \rfloor \ldots )</td>
</tr>
<tr>
<td>( &lt; )</td>
<td>( \ldots \lfloor \sup(E_i - 1)/n_i \rfloor )</td>
</tr>
<tr>
<td>( &gt; )</td>
<td>( \lfloor \inf(E_i + 1)/n_i \rfloor \ldots )</td>
</tr>
<tr>
<td>( \neq )</td>
<td>( \ldots \lfloor \sup(E_i)/n_i \rfloor \ldots \lfloor \inf(E_i)/n_i \rfloor )</td>
</tr>
</tbody>
</table>

Table 4.2: Translation of arithmetic constraints

The following lemma is immediate.

**Lemma 4.1** Given a store \( \sigma \) and a linear term \( t \). It follows that \( \inf(t)\sigma \leq t[y/n] \) and \( \sup(t)\sigma \geq t[y/n] \), for all \( n \in y^\sigma \).

**Proposition 4.2** Given a linear constraint \( c \), the indexicals \( F_c \), computed by Table 4.2, are a necessary translation of \( c \).

**Proof:** Let \( c \) be \( nx \leq E \) and \( \sigma \) be a store such that \( \sigma \) entails \( c \). It follows that \( m \leq \lfloor E[y/m]/n \rfloor \) for all \( m \in x^\sigma \) and \( m' \in y^\sigma \). Hence, by Lemma 4.1,
m \leq \lfloor \sup(E)_\sigma /n \rfloor$. Thereby, $x_\sigma \subseteq 0.\lfloor \sup(E)_\sigma /n \rfloor$. Similarly, it is proven that if $\sigma$ entails $nx \geq E$, then $x_\sigma \subseteq \lfloor \inf(E)_\sigma /n \rfloor$.

Since, $nx = E$ is entailed in $\sigma$ iff $nx \leq E$ and $nx \geq E$ are entailed in $\sigma$, the translation of $nx = E$ is necessary. Similarly, the translations for $nx \neq E$, $nx < E$ and $nx > E$ are necessary. \hfill \Box

Note that the proposed translation produces indexicals such that any pair of indexicals generated from an arithmetic constraint are equivalent, i.e. one of the indexicals is entailed iff the other one is. Furthermore, by Proposition 4.1 and 3.5, the indexicals are classified as monotone.

This compilation scheme has one major drawback: the code size is quadratic in the size of the input. This property is probably unacceptable except in toy programs or for binary/ternary constraints, and can be removed by using conjunctions of library calls instead (see Section 9.3). Another drawback, which is less obvious, is that the resulting code might recompute a lot of unchanged information each time it is reexecuted. Each time one of the $k$ indexicals is reexecuted due to a pruning of one of the right-hand side variables, $k = 1$ min or max attributes will be evaluated, even though possibly all but one of them are unchanged. However, it is possible to produce code in which only $O(\log k)$ min or max attributes are evaluated (see Section 9.3.1).

### 4.2 Necessary Translation of Disjunctions

We now extend the compilation of constraints into monotone indexicals to apply to any dnf constraint in FDC. Let $F_a$ denote the set of indexicals generated by compiling $a$ (see Section 4.1), where $a$ is an arithmetic constraint.

We proceed stepwise as follows:

1. Assume $c$ is equal to $c_1 \lor \ldots \lor c_n$, where each $c_i$ is a conjunction of arithmetic constraints.

2. Define $Y_i$ as $\{ \{ F_a : a \in c_i \} \}$ where any two indexicals $x$ in $r$ and $x$ in $r'$ have been replaced by $x$ in $r \land r'$. Hence, $Y_i = \{ y_1 \text{ in } r_1, \ldots, y_k \text{ in } r_k \}$, for some $l$ and $y_1, \ldots, y_k \in \{ x_1, \ldots, x_k \}$.

3. Let $V_i = \{ x_1, \ldots, x_k \} \setminus \{ y_1, \ldots, y_k \}$, and thus $X_i = Y_i \cup \{ x \text{ in } 0 : x \in V_i \}$.

Hence, $X_i$ is the result of translating conjunct $c_i$ to indexicals. We now turn to how the disjunction of $c_1, \ldots, c_n$ can be removed. Again, we proceed stepwise:

1. Let $X_i$ be as above, i.e. $X_i = \{ x_1 \text{ in } r_{i1}, \ldots, x_k \text{ in } r_{ik} \}$, $1 \leq i \leq n$.

We define $s_i$ as

$$s_i = (\text{dom}(x_1) \land r_{i1}) \Rightarrow \ldots \Rightarrow (\text{dom}(x_k) \land r_{ik}).$$
and \( r_i \) as
\[
  r_i = (s_1 \Rightarrow r_{1i}) \lor \ldots \lor (s_n \Rightarrow r_{ni}).
\]

2. Consequently, let \( F_c \) be the set \( \{ x_1 \text{ in } r_1, \ldots, x_k \text{ in } r_k \} \).

**Proposition 4.3** Let \( c \) be in dnf. Then \( F_c \) is a necessary translation of \( c \).

**Proof:** Let \( c(x_1, \ldots, x_k) \equiv c_1(x_1, \ldots, x_k) \lor \cdots \lor c_n(x_1, \ldots, x_k) \), and \( \sigma \) is a store which entails \( c \). Hence, \( \langle n_1, \ldots, n_k \rangle \in R_c \), where \( n_i \in x_{i\sigma} \). Thus, for some \( j \) between 1 and \( n \), \( \langle n_1, \ldots, n_k \rangle \in R_{c_j} \), where \( c_j \equiv a_1 \land \cdots \land a_m \) and \( a_i \) is an arithmetic constraint, \( 1 \leq l \leq m \). Thereby, for each \( l \), \( \langle n_1, \ldots, n_k \rangle \in R_{a_l} \).

By proposition 4.2, \( x_{i\sigma} \subseteq r_{j\sigma} \), where \( x_i \text{ in } r_{j\sigma} \in X_j \), for each \( i \) between 1 and \( k \). Hence, by definition, \( s_{j\sigma} \neq \emptyset \), i.e. \( x_{i\sigma} \subseteq r_{i\sigma} \). \( \square \)

Also, using Proposition 4.1, it follows that \( F_c \) consists of monotone indexicals, and that the monotonicity algorithm classifies them as monotone (Proposition 3.5).

When compiling arithmetic constraints, indexicals are generated which are equivalent to each other (Section 4.1). This can be used for optimizing the ranges generated by the compilation.

Let \( F_c \) be as above. \( F_c \) is optimized by removing redundant conditional ranges. There are several cases to apply:

- **\( s_i \)** For each \( s_i \), \( 1 \leq i \leq n \), replace any two ranges \( \text{dom}(x) \land r \) and \( \text{dom}(y) \land r' \), such that \( x \text{ in } r \) and \( y \text{ in } r' \) are generated from the same arithmetic constraint \( a \), by \( \text{dom}(x) \land r \).

- **\( r_j \)** For any indexical \( x_i \text{ in } r_0 \lor (\text{dom}(x) \land r) \Rightarrow r' \lor r_1 \), \( 1 \leq i \leq k \), replace \( (\text{dom}(x) \land r) \Rightarrow r' \) with \( r' \) if \( x_i \text{ in } r' \) and \( x \text{ in } r \) are generated from the same arithmetic constraint \( a \).

- **0.** A conditional range such as \( (\text{dom}(x) \land 0.) \Rightarrow r' \) is replaced by \( r' \), for any \( x \), since \( \text{dom}(x) \land 0. \) is nonempty in any consistent store.

In the examples below, the reductions are applied beforehand to reduce the complexity of the indexicals. There are other optimizations concerned with using intermediate variables for storing range value used multiple times, and for optimizing the evaluation of conditional ranges that we do not go into here.

Let us now consider a few examples of disjunctive constraints compiled into indexicals.

**Example 4.2:** Let \( c \) be the constraint \( x + i \leq y \lor y + j \leq x \), for some constants \( i \) and \( j \). The conjuncts are compiled into the two sets

\[
\{ x \text{ in } (\max(y) - i), y \text{ in } (\min(x) + i) \}, \text{ and }
\]
\[
\{ x \text{ in } (\min(y) + j), y \text{ in } (\max(x) - j) \}.
\]
These sets are conditioned and removed of redundant conditions computing $F_c$ as

$$
\begin{align*}
x \in . (\text{max}(y) - i) \lor (\text{min}(y) + j) . \\
y \in (\text{min}(x) + i) . \lor . (\text{max}(x) - j)
\end{align*}
$$

Note that $c$ is a typical scheduling constraint, stating that either $x$ precedes $y$ by some constant, or $y$ precedes $x$ by some constant. The intended behavior is to exploit the disjunction constructively, which the indexicals do. The domain of $x$ is effectively pruned of any number in the interval $[\text{max}(y) - i - 1, \text{min}(y) + j + 1]$, and similarly the domain of $y$ is pruned of any number in the interval $[\text{max}(x) - j - 1, \text{min}(x) + i + 1]$. This extends the pruning that is achieved by cardinality-based disjunction [HSD91, HSD92a].

**Example 4.2:** Let $c$ be the constraint $(x = 1 \land y = i_1) \lor (x = 2 \land y = i_2)$. The conjuncts are compiled into

$$
\{x \in 1, y \in \text{dom}(i_1)\} \text{ and } \{x \in 2, y \in \text{dom}(i_2)\},
$$

where the indexicals for $i_1$ and $i_2$ are ignored. Thus, $F_c =$

$$
\begin{align*}
x \in (\text{dom}(y) \land \text{dom}(i_1)) \Rightarrow 1) \lor ((\text{dom}(y) \land \text{dom}(i_2)) \Rightarrow 2), \\
y \in (\text{dom}(x) \land 1) \Rightarrow \text{dom}(i_1)) \lor ((\text{dom}(x) \land 2) \Rightarrow \text{dom}(i_2))
\end{align*}
$$

where no optimization rules are applicable.

Consider for a moment element $(x, l, y)$ which is true iff the $x$th element in $l$ is equal to $y$ [DSH88, CJH94]. The constraint is equivalent to

$$
\bigvee_j (x = j \land y = i_j)
$$

where $l = [i_1, \ldots, i_k]$ and $i_j$ is assumed to be determined, $1 \leq j \leq k$, of which $c$ is a particular case where $k = 2$. It has previously been shown that element/3 can be defined in terms of cardinality disjunction [HSD92b], however, under the assumption that $i_j$ is determined. Thus, our approach is slightly more general since $i_j$ need not be determined.

**Example 4.2:** Let $c$ be the constraint $x = z \lor y = z$. Hence, $F_c =$

$$
\begin{align*}
x \in ((\text{dom}(y) \land \text{dom}(z)) \Rightarrow 0.) \lor \text{dom}(z), \\
y \in ((\text{dom}(x) \land \text{dom}(z)) \Rightarrow 0.) \lor \text{dom}(z), \\
z \in \text{dom}(x) \lor \text{dom}(y)
\end{align*}
$$

which for example in the store $\{x \in \{1, 2\}, y \in \{3, \ldots, 6\}, z \in \{6\}\}$ propagates $y \in \{6\}$. Conjoining $F_c$ with $x \leq z$ and $y \leq z$ compiled to indexicals (see Section 4.1) thus gives a more powerful max/3 constraint than in clp(FD) [DC93b].

**Proposition 4.4** $F_c(\sigma) = \sigma$ does not imply that $c$ is arc-consistent in $\sigma$. 

4.3 Blocking Implication

Proof: A counterexample is the following:

Consider the disjunction

\[ c \equiv (x = y \land x = z \land y = 1) \lor (x = y \land x = z \land z = 1) \]

in

\[ \sigma = \{ x \in \{1, 2\}, y \in \{1, 2\}, z \in \{1, 2\} \}. \]

It follows that \( F_c(\sigma) = \sigma \), but applying \( c \) constructively in \( \sigma \) results in the store

\[ \sigma' = \{ x \in \{1\}, y \in \{1\}, z \in \{1\} \}. \]

Hence, by proposition 3.8, \( c \) is not arc-consistent in \( \sigma \).

4.3 Blocking Implication

The disjunctive normal form treats implication as non-blocking, i.e. \( c \rightarrow c' \) is interpreted as \( \neg c \lor c' \). However, we now show how the above scheme can be adapted to blocking implication as well, the intuition being that \( c' \) is not to be evaluated until \( c \) is entailed. In short, we transform the implication to a set of indexicals such that, by using the decision scheme of Table 3.1, the implication blocks the execution correctly.

Let \( F_c \) be the result of compiling \( \neg c \) as in Section 4.2, let \( s_c \) be computed from \( F_c \) as on page 50, and let \( \{ x_1 \text{ in } r_1, \ldots, x_k \text{ in } r_k \} \) be the result of compiling \( c' \) as in Section 4.2.

Hence, compiling \( c \rightarrow c' \) generates the set of indexicals

\[ b \text{ in } s_c \Rightarrow 0 \lor 1, \]

and

\[ x_1 \text{ in } (\max(b) \ldots \min(b)) \Rightarrow r_1, \ldots, x_k \text{ in } (\max(b) \ldots \min(b)) \Rightarrow r_k \]

i.e. as long as \( s_c \) evaluates to a nonempty set, \( b \) is not determined and \( (\max(b) \ldots \min(b)) \Rightarrow r_i \) is not monotone. However, whenever \( s_c \) evaluates to empty, \( b \) is determined to 1, and \( x_i \text{ in } (\max(b) \ldots \min(b)) \Rightarrow r_i \) is evaluated, 1 \( \leq i \leq k \). That is, if \( \neg c \) is inconsistent, \( s_c \) evaluates to empty, and thus when \( c \) is entailed \( x_i \) is constrained by \( r_i \), i.e. \( c' \) is evaluated as soon as \( c \) is entailed.

Note further that the generated indexicals for implications can be optimized in many cases. For example, see Section 8.5.3 where a somewhat different formulation of \( b = 1 \Leftrightarrow x = v \) than what is generated above is used, which is optimized by combining several indexicals into one.
4.4 Arithmetics Revisited

4.4.1 Nonlinear constraints

In Section 4.1 we only considered linear arithmetic constraints. Having introduced blocking implication we now consider extending the compilation method to nonlinear constraints. Blocking implication is used for avoiding division by 0, which is a problem that otherwise would occur.

Let \( \inf \) (\( \sup \)) be as before, i.e. \( \inf(t) \) (\( \sup(t) \)) is the smallest (largest) value that \( t \) can ever get (see Table 4.1). The functions are extended to nonlinear terms as shown in Table 4.3. The table assumes that \( x \) and \( y \) range over natural numbers. Obviously, Lemma 4.1 still applies to the extended \( \inf \) and \( \sup \) functions.

<table>
<thead>
<tr>
<th>( t )</th>
<th>( \inf(t) )</th>
<th>( \sup(t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( xy )</td>
<td>( \inf(x) * \inf(y) )</td>
<td>( \sup(x) * \sup(y) )</td>
</tr>
</tbody>
</table>

Table 4.3: Upper and lower bounds of nonlinear products

4.4.2 A necessary translation

The method is best explained by an example.

Example 4.4.2: The constraint

\[
xy \leq 5
\]

is approximated by the indexicals

\[
\begin{align*}
b_y & \text{ in } (\text{dom}(y) \land 0) \Rightarrow 0 \lor 1, \\
b_x & \text{ in } (\text{dom}(x) \land 0) \Rightarrow 0 \lor 1, \\
x & \text{ in } (\text{max}(b_y).\text{min}(b_y)) \Rightarrow 0.\lfloor 5/\text{min}(y) \rfloor, \\
y & \text{ in } (\text{max}(b_x).\text{min}(b_x)) \Rightarrow 0.\lfloor 5/\text{min}(x) \rfloor,
\end{align*}
\]

The compilation of inequations and disequations is completely analogous to that of equations. In general, the idea is to rewrite a constraint over the variables \( x_1 \ldots x_k \) into the equivalent constraints

\[
\begin{align*}
n_1x_1y_1 & \cdot E_1 \\
& \vdots \\
n_kx_ky_k & \cdot E_k
\end{align*}
\]
where \( \cdot \) is the relation symbol and then to translate them into a conjunction of indexicals

\[
\begin{align*}
& b_{y_1} \text{ in } (\text{dom}(y_1) \land 0) \Rightarrow 0 \lor 1, \\
& b_{x_1} \text{ in } (\text{dom}(x_1) \land 0) \Rightarrow 0 \lor 1, \\
& x_1 \text{ in } (\max(b_{y_1}) \cdot \min(b_{y_1})) \Rightarrow r_1, \\
& y_1 \text{ in } (\max(b_{x_1}) \cdot \min(b_{x_1})) \Rightarrow r'_1, \\
& \vdots \\
& b_{y_k} \text{ in } (\text{dom}(y_k) \land 0) \Rightarrow 0 \lor 1, \\
& b_{x_k} \text{ in } (\text{dom}(x_k) \land 0) \Rightarrow 0 \lor 1, \\
& x_k \text{ in } (\max(b_{y_k}) \cdot \min(b_{y_k})) \Rightarrow r_k, \\
& y_k \text{ in } (\max(b_{x_k}) \cdot \min(b_{x_k})) \Rightarrow r'_k,
\end{align*}
\]

that propagate information whenever the \( \min \) or the \( \max \) of a variable changes, where \( r_i \) (\( r'_i \)) is defined from \( E_i \), \( n_i \) and \( y_i \) (\( x_i \)) (see Table 4.4).

<table>
<thead>
<tr>
<th>( \cdot )</th>
<th>( r_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( = )</td>
<td>([\inf(E_i)/n_i \cdot \max(y_i)] \cdot [\sup(E_i)/n_i \cdot \min(y_i)])</td>
</tr>
<tr>
<td>( \leq )</td>
<td>([\sup(E_i)/n_i \cdot \min(y_i)])</td>
</tr>
<tr>
<td>( \geq )</td>
<td>([\inf(E_i)/n_i \cdot \max(y_i)])</td>
</tr>
<tr>
<td>( &lt; )</td>
<td>([\sup(E_i - 1)/n_i \cdot \min(y_i)])</td>
</tr>
<tr>
<td>( &gt; )</td>
<td>([\inf(E_i + 1)/n_i \cdot \max(y_i)])</td>
</tr>
<tr>
<td>( \neq )</td>
<td>(-[\sup(E_i)/n_i \cdot \min(y_i)] \cdot [\inf(E_i)/n_i \cdot \max(y_i)])</td>
</tr>
</tbody>
</table>

Table 4.4: Translation of consistency constraints

**Proposition 4.5** Given a nonlinear constraint \( c \), the indexicals \( F_c \) computed by Table 4.2, are a necessary translation.

**Proof:** Consider \( c \equiv nxy \leq E \), and let \( \sigma \) be a store that entails \( c \). Hence, \( mm' \leq |E[z/m'/n]| \), where \( m \in x_\sigma \), \( m' \in y_\sigma \), and \( m'' \in z_\sigma \). Thus, if \( m \neq 0 \), \( m' \leq |E[z/m'/n]| \), i.e. by Lemma 4.1, \( m' \leq |\sup(E)\sigma/n \cdot \min(x)\sigma| \).

Also, \((\text{dom}(x) \land 0) \Rightarrow 0 \lor 1)\sigma = \{1\} \) iff \( 0 \not\in x_\sigma \). Thus, \( b_\sigma = \{1\} \) iff \( 0 \not\in x_\sigma \), and otherwise \( b_\sigma = \{0,1\} \). Thereby, \((\max(b) \cdot \min(b))\sigma \neq \emptyset \) iff \( 0 \not\in x_\sigma \). It follows that

\[
y_\sigma \subseteq ((\max(b) \cdot \min(b)) \Rightarrow [\sup(E)\sigma/n \cdot \min(x)\sigma].
\]

Similar reasoning applies to \( x_\sigma \), and to \( nxy \geq E \). The cases for \( nxy = E \), \( nxy > E \), \( nxy < E \), and \( nxy \neq E \) are proven as in Proposition 4.2. \( \Box \)
An indexical $b \in (\text{dom}(y) \land \neg 0) \Rightarrow 0 \lor 1$ is classified as monotone (Proposition 3.5), and the indexical $x \in (\text{max}(b) \cdot \text{min}(b)) \Rightarrow r$ is classified as monotone in any store where $b$ is determined, which follows by Proposition 3.5 and 4.1.

This translation scheme suffers from the same drawbacks as the inline compilation of linear arithmetic does (Section 4.1.1), hence we also consider the extension of the library method to nonlinear constraints (Chapter 9.3.2).

An obvious optimization is to remove the conditional ranges at compile time when it can be inferred that division by 0 cannot occur. This optimization applies to any of the following constraints:

$$nxy : E$$

where $\cdot \in \{=, >, \geq\}$, $n$ is a constant, $x$ and $y$ are finite domain variables, and $E = n_1x_1y_1 + \ldots + n_kx_ky_k + n_0$, where $n_i$ is a constant, $n_0 > 0$, $x_i$ and $y_i$ are finite domain variables ($0 \leq i \leq k$). Thus, instead the following indexicals are computed:

$$x \in r,$$
$$y \in r',$$
$$x \in 1..,$$
$$y \in 1..$$

where $r$ ($r'$) is defined from $E$, $n$ and $y$ ($x$) (see Table 4.4).

### 4.4.3 Full arc-consistency of linear (dis)equalities

Consider a linear (dis)equality $c$, translated into $F_c$ as in Section 4.1.1. Because of the interval reasoning used, it is not true that for each $\sigma$, $\sigma$ is a fixed point of $F_c$ iff $c$ is arc-consistent in $\sigma$.

However, by using range expressions of the form $r + r'$ we can in fact give a necessary translation such that arc-consistency is preserved by the evaluation of the indexicals. We explain the idea with an example.

**Example 4.4.3:** The constraint

$$2x = 3y + 5z$$

is rewritten thrice to equivalent equations, each expressing a single variable as a function of the others:

$$2x = 3y + 5z,$$
$$3y = 2x - 5z,$$
$$5z = 2x - 3y.$$
These equations are approximated by the indexicals:

\[
\begin{align*}
  x \text{ in } & \left[ 3 \ast \text{dom}(y) + 5 \ast \text{dom}(z)/2 \right] \land \left[ 3 \ast \text{dom}(y) + 5 \ast \text{dom}(z)/2 \right], \\
y \text{ in } & \left[ 2 \ast \text{dom}(x) - 5 \ast \text{dom}(z)/3 \right] \land \left[ 2 \ast \text{dom}(x) - 5 \ast \text{dom}(z)/3 \right], \\
z \text{ in } & \left[ 2 \ast \text{dom}(x) - 3 \ast \text{dom}(y)/5 \right] \land \left[ 2 \ast \text{dom}(x) - 3 \ast \text{dom}(y)/5 \right].
\end{align*}
\]

\[
\square
\]

**Proposition 4.6** Given \( c \equiv nx = E \) (\( nx \neq E \), \( nxy = E \), \( nxy \neq E \)), the indexical \( f \equiv x \text{ in } \left[ E[y/\text{dom}(y)]/n \right] \land \left[ E[y/\text{dom}(y)]/n \right] \) is a necessary translation. Furthermore, \( \sigma \) is a fixed point of \( f \) iff \( c \) is arc-consistent in \( \sigma \).

**Proof:** Suppose \( \sigma \) entails \( c \). Hence, \( nm = E[y/m'] \), where \( m \in x_\sigma \) and \( m' \in y_\sigma \). Thus, \( \left[ E[y/m']/n \right] \leq m \leq \left[ E[y/m']/n \right] \). It follows that

\[
m \in (\left[ E[y/\text{dom}(y)]/n \right]_\sigma \land m \in (\left[ E[y/\text{dom}(y)]/n \right]_\sigma ,
\]

i.e. \( x_\sigma \subseteq (\left[ E[y/\text{dom}(y)]/n \right] \land \left[ E[y/\text{dom}(y)]/n \right]_\sigma \).

In the following we only consider \( x \), however, the same reasoning can be applied to any other variable \( y \) in \( E \) by rewriting \( nx = E \) into \( n'y = E' \). Thus, \( c \) arc-consistent in \( \sigma \) iff for all \( m \in x_\sigma \), there exists \( m' \in y_\sigma \), for all other variables \( y \) in \( E \), such that \( nm = E[y/m'] \), which is true iff \( \left[ E[y/m']/n \right] \leq m \leq \left[ E[y/m']/n \right] \), i.e. \( x_\sigma \subseteq (\left[ E[y/\text{dom}(y)]/n \right] \land \left[ E[y/\text{dom}(y)]/n \right]_\sigma \).

Similar reasoning proves the cases of \( nx \neq E \), \( nxy = E \), and \( nxy \neq E \). \( \square \)

Note that by Complexity 3.1, the complexity of evaluating the expression \( \left[ E[y/\text{dom}(y)]/n \right] \land \left[ E[y/\text{dom}(y)]/n \right] \) is strictly higher than evaluating \( \text{inf}(E)/n \land \text{sup}(E)/n \) if there is more than one variable \( y \) in \( E \).

### 4.5 Conclusion

The indexical scheme for compiling constraints in FDC is based solely on indexicals and their evaluation as defined by Table 3.1, while maintaining the constructive reading of disjunctions, and the blocking aspect of implication. Furthermore, we treat arithmetic constraints, linear and nonlinear such, by transformation into indexicals, thereby preserving a necessary translation. The translation preserves partial arc-consistency, but can be extended to full arc-consistency for linear constraints.
Chapter 5

Entailment Checking

We now consider entailment checking of constraints in FDC. This is done by compiling a constraint $c$ in FDC into an antimonotone indexical which denotes a truth condition of $c$. Hence, the translation is sufficient (Section 3.8).

5.1 Linear Arithmetic Constraints

The compilation of an arithmetic constraint $c$ used for entailment checking is based on deriving a sufficient condition $S_c$ for $c$ such that $S_c$ entails $c$. Having obtained $S_c$, we express it as a single antimonotone indexical.

Consider the example

$$c \equiv 2x + 3y \geq 5.$$  

A suitable $S_c$ is obtained by replacing $2x + 3y$ by its lower bound. Thus, we obtain

$$S_c \equiv 2 \min(x) + 3 \min(y) \geq 5$$

and its formulation as a single antimonotone indexical

$$5 \text{ in } (2 \min(x) + 3 \min(y)).$$

Hence, the compilation is quite simple: after normalization to $S + i \cdot T + j$ (Section 3.6.1) such that $i \geq j$, the constraint $i - j \cdot T - S$ is translated into the corresponding antimonotone indexical $i - j \text{ in } r$ where $r$ is defined by the constraint predicate (see Table 5.1).

The derivation of these translation rules is completely analogous to that of the rules compiling consistency constraints (see Table 4.2), except that sup and inf are swapped to obtain sufficient conditions instead of necessary conditions for the constraint: a sufficient condition for $x \leq E$ is the indexical $x \text{ in } \inf(E)$; a sufficient condition for $x \geq E$ is the indexical $x \text{ in } \sup(E)$;
\[ \begin{array}{|c|c|} \hline & r \\ \hline \leq & \sup(T - S) \cdot \inf(T - S) \\ \geq & \inf(T - S) \\ \sup & \sup(T - S) \\ \inf & \sup(T - S - 1) \\ < & \sup(T - S + 1) \\ > & -\inf(T - S) \cdot \sup(T - S) \\ \hline \end{array} \]

Table 5.1: Translation of entailment constraints

\(\land\) and \(\lor\) are expressed as intersection and union; and all six relations can be expressed in terms of \(\leq, \geq, \land, \lor\).

**Proposition 5.1** The translation of a linear constraint \(c\) into the indexical \(f\) (see Table 5.1) is sufficient.

**Proof:** Consider \(i - j \leq T - S\). Hence, \(f \equiv i - j \in 0 \cdot \inf(T - S)\). Suppose \(\sigma\) is a fixed point of \(f\). Hence, \(0 \leq i - j \leq \inf(T - S)_\sigma\). By Lemma 4.1, \(i - j \leq (T - S)[y/m]\), for all \(m \in y_e\), i.e. \(\sigma\) entails \(i - j \leq T - S\). Similar reasoning proves the case for \(i - j \geq T - S\). Hence, \(i - j \in \sup(T - S) \cdot \inf(T - S)\) is sufficient for \(i - j = T - S\). The cases for \(i - j \neq T - S\), \(i - j > T - S\), and \(i - j < T - S\) are shown analogously.

Since the size of the indexical is linear in the size of the constraint, there is no need to consider compilation to library constraints. However, for the sake of Section 5.3 we normalize \(S + i \cdot T + j\) into \(S + i - T - j \cdot 0\) instead, thus generating an indexical \(0 \in r\). Also, using Proposition 4.1 it follows that \(r\) is classified as antimonotone.

### 5.2 Nonlinear Arithmetic Constraints

The compilation of nonlinear constraints can proceed precisely as for linear constraints (Section 5.1).  

**Example 5.2:** Consider

\[ c \equiv xy \geq 5. \]

A suitable truth condition \(S_c\) is obtained by replacing \(xy\) by its lower bound. Thus, we obtain

\[ S_c \equiv \min(x) \cdot \min(y) \geq 5 \]

and its formulation as a single antimonotone indexical

\[ 5 \in (\min(x) \cdot \min(y)). \]

Hence, after normalization to \(S + i \cdot T + j\) such that \(i \geq j\), the constraint

\[ i - j \cdot T - S \]

is translated into the corresponding antimonotone indexical

\[ i - j \in r \] where \(r\) is defined by the constraint predicate (see Table 5.1).
Proposition 5.2 The translation of a nonlinear constraint \( c \) into the indexical \( f \) (see Table 5.1) is sufficient.

Proof: The proof is exactly as for linear constraints. \( \square \)

As for linear constraints, we consider the translation of \( S + i - T - j \cdot 0 \) in Section 5.3. Also, using Proposition 4.1 it follows that \( r \) is classified as antimonotone.

5.3 Constraints in Disjunctive Normal Form

Let \( c \) be a dnf constraint in FDC used for entailment checking, i.e. the entailment of \( c \) is needed for some computation to proceed.

We thus translate \( c \) into an antimonotone indexical which denotes a sufficient condition on the store to entail \( c \), i.e. when the indexical is entailed, detected as by Table 3.1, so is \( c \). However, \( c \) may be entailed in a weaker store than the indexical. Experiments have shown that the indexical is weak enough for practical purposes [CJH94].

Since \( c \) is in dnf it follows that \( c \equiv c_1 \lor \ldots \lor c_n \), where each \( c_i \) is a conjunction of arithmetic constraints. Define \( Y_i = \{ 0 \in r_i : a \in c_i \} \), where \( 0 \in r_a \) is the indexical generated by compiling \( a \) for entailment (Section 5.1), \( 1 \leq i \leq n \).

Replace each pair \( 0 \in r \) and \( 0 \in r' \) in \( Y_i \) by \( 0 \in r \wedge r' \), thus resulting in a singleton set \( X_i = \{ 0 \in r_i \} \).

Finally, given \( X_1, \ldots, X_n \) derive \( 0 \in r \), where \( r = r_1 \lor \ldots \lor r_n \).

Proposition 5.3 The translation of a constraint \( c \) in dnf into the indexical \( f \) (see Table 5.1) is sufficient.

Proof: Let \( c \equiv c_1 \lor \ldots \lor c_n \). Suppose \( \sigma \) is a fixed point of \( f \). Hence, \( 0 \in r_1, \ldots, r_n \), i.e. for some \( j \), \( 0 \in r_j \), where \( r_j \equiv r_1 \land \ldots \land r_n \). Thereby, for each \( l \) between 1 and \( n \), \( 0 \in r_{n, \sigma} \). By propositions 5.1 and 5.2, \( \sigma \) entails \( a_l \), and, hence, \( \sigma \) entails \( c_j \), i.e. \( \sigma \) entails \( c \). \( \square \)

Using Proposition 4.1 and 3.5 it follows that \( r \) is classified as antimonotone.

5.3.1 Some examples

We consider again the constraints in Section 4.2, but this time we compile them for entailment checking.

Example 5.3.1: Let \( c \) be the constraint \( x > 1 \rightarrow y = 2 \). The compilation proceeds as follows.

\( c \) is rewritten into dnf as

\[ x \leq 1 \lor y = 2, \]
where the conjuncts are compiled into

\[
\begin{align*}
0 & \text{ in } (\max(x) - 1), \\
0 & \text{ in } (\max(y) - 2) \cdot (\min(y) - 2)
\end{align*}
\]

respectively, which are combined as

\[
0 \text{ in } (\max(x) - 1) \lor (\max(y) - 2) \cdot (\min(y) - 2),
\]

which is entailed when \(x\) is less or equal to 1, or when \(y\) is equal to 2.

**Example 5.3.1:** Let \(c\) be the constraint \(x + i \leq y \lor y + j \leq x\), for some constants \(i\) and \(j\). The compilation proceeds as follows.

\(c\) is already in dnf, so the conjuncts are compiled into

\[
\begin{align*}
0 & \text{ in } (\min(y) - i - \max(x)) \\
0 & \text{ in } (\min(x) - j - \max(y))
\end{align*}
\]

respectively, which are combined as

\[
0 \text{ in } (\min(y) - i - \max(x)) \lor (\min(x) - j - \max(y)),
\]

which is entailed when \(x\) is less or equal to \(y - i\), or when \(y\) is less or equal to \(x - j\).

**Example 5.3.1:** Let \(c\) be the constraint \((x = 1 \land y = i_1) \lor (x = 2 \land y = i_2)\), for some constants \(i_1\) and \(i_2\). The compilation proceeds as follows.

\(c\) is already in dnf, so the conjuncts are compiled into

\[
\begin{align*}
0 & \text{ in } (\max(x) - 1) \cdot (\min(x) - 1) \land (\max(y) - i_1) \cdot (\min(y) - i_1) \\
0 & \text{ in } (\max(x) - 2) \cdot (\min(x) - 2) \land (\max(y) - i_2) \cdot (\min(y) - i_2)
\end{align*}
\]

respectively, which are combined as

\[
0 \text{ in } ((\max(x) - 1) \cdot (\min(x) - 1) \land (\max(y) - i_1) \cdot (\min(y) - i_1)) \lor ((\max(x) - 2) \cdot (\min(x) - 2) \land (\max(y) - i_2) \cdot (\min(y) - i_2)),
\]

which is entailed when \(x\) is equal to 1 and \(y\) is equal to \(i_1\), or \(x\) is equal to 2 and \(y\) is equal to \(i_2\).

### 5.4 Implication

Compiling \(c \rightarrow c'\) for entailment is either done by compiling \(\neg c \lor c'\) for entailment, or as follows, where the difference is that the following translation only succeeds when the entailment of \(c\) implies the entailment of \(c'\).

Let \(0 \text{ in } r\) be the indexical generated by compiling \(c'\) for entailment, and let \(s_c\) be as in Section 4.3. Thus, \(c \rightarrow c'\) is compiled into

\[
\{b \text{ in } s_c \Rightarrow 0 \lor 1, 0 \text{ in } (\min(b) \cdot \max(b)) \Rightarrow r\},
\]

i.e. while \(s_c\) is true, \(b\) is not determined. Therefore, when \(c\) is true, \(b\) is determined. Thereby, \(0 \text{ in } r\) becomes antimonotone and can be used for checking the entailment of \(c'\).
5.5 Symbolic Constraints

We now show how the entailment of atmost, atleast, and element can be checked.

5.5.1 element

Consider again the definition of element as in Section 3.6.7. The constraint $\text{element}(i, [x_1, \ldots, x_k], v)$ is true if $c \equiv \bigvee_{j=1}^{k} (i = j \land x_j = v)$ is true, where we assume $x_j$ is a constant, $1 \leq j \leq k$. Hence, we compile $c$ for entailment to check the entailment of $\text{element}(i, [x_1, \ldots, x_k], v)$.

5.5.2 atmost/atleast

Consider the constraint atmost$(u, l, v)$ which is true iff at most $u$ elements in $l$ are equal to $v$, where $l = [x_1, \ldots, x_k]$. The entailment of the constraint can be checked by the formula

$$\sum_{i=1}^{k} (b)_i \leq u$$

where $(b)_i$ is 1 if $x_i = v$ is true and 0 if $x_i \neq v$ is true.

atleast$(u, l, v)$ is similarly defined as

$$\sum_{i=1}^{k} (b)_i \geq u$$

5.6 Conclusion

We show how the entailment of any constraint $c$ in FDC can be checked by a corresponding antimonotone indexical denoting a nontrivial sufficient condition on $c$. Given our constraint solver (Chapter 6) which treats monotone as well as antimonotone indexicals, we can thus efficiently check the entailment of any constraint in FDC.
Chapter 6

Implementation of FD

The implementation of FD is described in detail. We take care to present the implementation separate from AKL specifics. In short, the chapter contains the following: the FD solver with its optimizations, the algorithm for computing monotonicity of ranges, a description of what information must be kept associated with finite domain variables and indexicals in order to realize all necessary reasoning and optimizations, the range emulator, and the range compiler with its optimizations.

6.1 The Basic FD Solver

First we need to define how an indexical is suspended. Given an indexical \( x \) in \( r \), by linking each domain variable \( y \) that occurs in \( r \) to (a representation of) the indexical, we say that the indexical is suspended (on \( y \)).

We now give an immediate formulation of Table 3.1 as a solver algorithm for indexicals. Hence, the algorithm checks the entailment/inconsistency of indexicals in a store \( \sigma \). A necessary, but insufficient, condition on when an indexical \( f \) is entailed in \( \sigma \) is that \( \sigma \) is a fixed point of \( f \).

**Fixed-point Condition:** Suppose \( f(\sigma) \neq \sigma \). If further \( f \) is monotone, then \( f(\sigma) \) is added to \( \sigma \) by the solver, since for any \( \sigma' \), such that \( \sigma \subseteq \sigma' \) and \( \sigma' \) entails \( f \), \( f(\sigma) \sqcup \sigma \subseteq \sigma' \) by the monotonicity of \( f \). Thereby, the FD solver is a fixed point algorithm for a set of indexicals \( F \) evaluated in a store \( \sigma \), which terminates when for each monotone \( f \) in \( F \), \( \sigma \) has been updated to a fixed point of \( f \), or to be inconsistent with \( f \).

We give two alternative formulations of the unoptimized solver, with the same complexity, that differ in what the propagation queue consists of.

6.1.1 Queue of variables

Let \( Q \) be a finite queue/set of variables, and let \( \sigma \) be a constraint store.
Algorithm 6.1.1:

variable check(Q, σ)
{
    while Q not empty do {
        set Q = Q \ {y};
        let F be the set of indexicals suspended on y;
        for each x in r in F do {
            let I = x_σ \ r_σ;
            case I of {
                ∅:
                    if r monotone in σ then fail else continue;
                x_σ:
                    if r antimonotone in σ then dismiss the indexical;
                    continue;
                otherwise:
                    if r not monotone in σ then continue;
                    set σ = σ \ {x ∈ I} and Q = Q \ {x};
            }
        }
    }
    return σ;
}

Hence, the outermost iteration is done over a queue of variables.

This is a naïve algorithm which can be much improved (see next section).
Its complexity depends on the number n of variables, the maximum domain size m of any variable, the maximum number e of suspended indexicals of any variable, and the cost c of evaluating a range.

Complexity 6.1 The solver terminates in $O(mnec)$ time.

Proof: The size of the variable queue is $O(n)$. For each dequeueing, at most $e$ indexicals are evaluated. A variable can only be pruned $m-1$ times, i.e. it can be dequeued at most $m$ times, thus the total number of indexical evaluations is less or equal to $mne$. Each indexical evaluation is $O(c)$ in time, i.e. the algorithm is $O(mnec)$ in time.

Note that, by the arc-consistency preserving translation in Section 4.4.3, each k-ary arithmetic constraint in FDC is translated into k indexicals, where for each such indexical x in r, r contains k range additions (subtractions), where each atomic subrange of r is a dom(y) expression for some y. Hence, c above equals $m^k$, and thereby the complexity of Algorithm 6.1.1 is $O(m^{k+1}ne)$, i.e. it coincides with Algorithm 3.9.1.

In Section 6.3, we give the optimized version of the algorithm, which has better average-case complexity.
6.1.2 Queue of indexicals

The alternative algorithm uses a queue of indexicals instead. Let \( F \) be a finite queue/set of (suspended) indexicals, and let \( \sigma \) be a constraint store.

**Algorithm 6.1.2:**

```plaintext
indexical_check(\( F \), \( \sigma \))
{
    while \( F \) not empty do {
        set \( F = F \setminus \{x \in r\} \);
        let \( I = x_\sigma \cap r_\sigma \);
        case \( I \) of {
            \emptyset:
                  if \( r \) monotone in \( \sigma \) then fail else continue;
            \( x_\sigma \):
                  if \( r \) antimonotone in \( \sigma \) then dismiss the indexical;
                  continue;
            otherwise:
                  if \( r \) not monotone in \( \sigma \) then continue;
                  let \( F_x \) be the set of indexicals suspended on \( x \);
                  set \( \sigma = \sigma \cup \{x \in I\} \) and \( F = F \cup F_x \);
        }
    }
    return \( \sigma \);
}
```

The complexity of the `indexical_check` algorithm depends on the number \( n \) of variables, the maximum domain size \( m \) of any variable, the maximum number \( e \) of suspended indexicals of any variable, and the cost \( c \) of evaluating a range.

**Complexity 6.2**  
The algorithm is \( O(mne \sigma) \) in time.

**Proof:** The size of the indexical queue is bounded by the number of indexicals, i.e. by \( ne \). A variable can be pruned at most \( m - 1 \) times, each time adding at most \( e \) indexicals to the queue, i.e. the number of queue updates is less or equal to \( \Sigma_{i=1}^{n} (m - 1) e = n(m - 1)e \). Hence, the number of times an indexical is evaluated is less or equal to \( ne + (m - 1)ne = mne \), and, thereby, the algorithm is \( O(mne) \) in time.

Which algorithm to use as the solver kernel is a matter of choice. The first algorithm is our choice, and we will next explain crucial optimizations which improve the efficiency of the algorithm by avoiding useless evaluations of ranges.

6.2 Solver Optimizations for FD

The optimizations that we now consider are concerned with improving the behavior of the basic FD solver (see Algorithm 6.1.1), such that indexicals
are evaluated as few times as possible.

**Variable dependencies**

This optimization is standard in finite domain systems [Hen89, AB91, HDT92, DC93c].

Consider the indexical $x \text{ in } 0 \cdot \max(y)$, and the store $\sigma = \{ x \in \{1, 2, 3\}, y \in \{1, 2, 3\} \}$ which is a fixed point of the indexical. Now suppose $\sigma$ is updated to $\sigma' = \{ x \in \{1, 2, 3\}, y \in \{2, 3\} \}$. Since $\sigma'$ is also a fixed point of the indexical, $x \text{ in } 0 \cdot \max(y)$ should not be evaluated. This is solved by registering the dependency between the variables in the range and the indexical.

The **dependency** between a variable $y$ occurring in a range $r$, and $r$ is defined by the following rules:

- If $y$ occurs as a term in $r$, the dependency always equals INT. Otherwise;
- If $y$ occurs as $\min(y)$ (or $\max(y)$) in $r$ but not both, the dependency equals MIN (or MAX);
- If $y$ occurs as $\min(y)$ and $\max(y)$ in $r$, the dependency equals $\min$-$\max$;
- If $y$ occurs only as $\text{dom}(y)$ in $r$, the dependency equals DOM.

Given a store $\sigma$ and a domain variable $x$. Then, a domain constraint $x \in I$, such that $x_\sigma \cap I \subset x_\sigma$, prunes $x$ (in $\sigma$).

Hence, the optimization we use states that, given an indexical $x \text{ in } r(y)$ in a store $\sigma$ where $y$ is pruned by $y \in I$, the indexical is to be evaluated only if the dependency between $y$ and $r$ is affected by the pruning (see Algorithm 6.2). For example, if $\min(y_\sigma \cap I) > \min(y_\sigma)$, any MIN or MINMAX dependency is affected.

If $y_\sigma \cap I$ is a singleton, indexicals previously monotone (antimonotone) may then become constant. Due to the incompleteness of our FD solver, any dependency is thereby said to be affected.

**Example 6.2:** Consider $f \equiv 0 \text{ in } 1 \cdot \min(y)$ and the store $\sigma = \{ y \in \{0, 1\} \}$, i.e. $f$ is inconsistent in $\sigma$ but since $f$ is antimonotone the solver will not detect this. If $\sigma$ is updated to $\{ y \in \{0\} \}$, $f$ becomes constant and thus the inconsistency is detected.

Similarly, consider $f \equiv x \text{ in } \min(y)$, and the store $\sigma = \{ x, y \in \{0, 1\} \}$, i.e. $\sigma$ is a fixed point of $f$, but does not entail $f$. If $\sigma$ is updated to $\{ y \in \{0\} \}$, the store is still a fixed point of $f$, however, $f$ is then also entailed.

**Algorithm 6.2:**

```plaintext
affected(y, I, \sigma)
{
    let yold = y_\sigma and ynew = y_\sigma \cap I;
```
if \(|y_{\text{new}}| = 1\) then return TRUE;
if \(\min(y_{\text{new}}) > \min(y_{\text{old}})\) then \(p_n = 1\) else \(p_n = 0\);
if \(\max(y_{\text{new}}) < \max(y_{\text{old}})\) then \(p_x = 1\) else \(p_x = 0\);
case dependency between \(y\) and \(r\) of {DOM:
    return TRUE;
    MIN:
        if \(p_n\) return TRUE; else return FALSE;
    MAX:
        if \(p_x\) return TRUE; else return FALSE;
    MINMAX:
        if \(p_n \lor p_x\) return TRUE; else return FALSE;
}

Therefore, the dependency between a variable and an indexical is registered in the suspension link, and the type of the pruning is associated with each variable in the propagation queue in Algorithm 6.1.1.

We use in the following DOM to denote the case when \(\min(y_{\text{new}}) = \min(y_{\text{old}}) \land \max(y_{\text{new}}) = \max(y_{\text{old}})\) is true, MIN to denote \(\min(y_{\text{new}}) > \min(y_{\text{old}})\), MAX to denote \(\max(y_{\text{new}}) < \max(y_{\text{old}})\), and MINMAX the case when \(\min(y_{\text{new}}) > \min(y_{\text{old}}) \land \max(y_{\text{new}}) < \max(y_{\text{old}})\) is true.

**Entailment marking**

This optimization concerns distributing information of what indexicals are entailed, so that certain indexicals can be ignored in the propagation.

Indexicals, known to be logically equivalent, are connected by references to a common flag (Section 6.4.2). Whenever one of the indexicals is decided entailed it is marked entailed by setting the flag. Hence, before any indexical is executed its entailment flag is checked and, if set, the indexical is ignored and can be dismissed.

The information that two (or more) indexicals are equivalent can be generated by the compiler, which compiles arithmetic constraints into conjunctions of equivalent indexicals (Section 9.3), or by the user who can annotate the occurrences of the indexicals.

**Propagation queues are sets**

A variable may be pruned one or several times within one activation of variable check. The queue should represent a set, avoiding the duplication of indexical executions. Therefore, at most one instance of each variable must be in the queue at any time.

This is solved by using a link field in each variable by which the variable is queued, and which can be used for testing whether the variable is queued. However, if queued, the previously registered pruning type of the variable
must be updated as in Table 6.1, where P and P’ are two pruning types that are combined, and the result is shown in the table. Consequently, no extra space is needed to allocate the queue.

<table>
<thead>
<tr>
<th>Result</th>
<th>P=DOM</th>
<th>P=MIN</th>
<th>P=MAX</th>
<th>P=MINMAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>P'=DOM</td>
<td>DOM</td>
<td>MIN</td>
<td>MAX</td>
<td>MINMAX</td>
</tr>
<tr>
<td>P'=MIN</td>
<td>MIN</td>
<td>MIN</td>
<td>MINMAX</td>
<td>MINMAX</td>
</tr>
<tr>
<td>P'=MAX</td>
<td>MAX</td>
<td>MINMAX</td>
<td>MAX</td>
<td>MINMAX</td>
</tr>
<tr>
<td>P'=MINMAX</td>
<td>MINMAX</td>
<td>MINMAX</td>
<td>MINMAX</td>
<td>MINMAX</td>
</tr>
</tbody>
</table>

Table 6.1: Pruning update

**Time stamps**

A negative consequence of using a queue of variables, instead of a queue of indexicals, is that multiple occurrences of indexicals in the queue are not filtered out. However, there is a simple cure for this.

To each indexical and variable, associate a time stamp. Add a time register \( T \) to Algorithm 6.1.1. Before queueing a variable, \( T \) is incremented, and the variable is stamped with the new value of \( T \). After an indexical is executed, it is stamped with the current value of \( T \). Hence, before executing an indexical \( f \), suspended on a variable \( x \), say, if the stamp of \( f \) is larger or equal than the stamp of \( x \), ignore \( f \). Initially, \( T \) and all stamps equal 0.

This optimization is also exploited in clp(FD) [DC93c].

**Propagation between equivalent indexicals**

The following optimization is exploited in clp(FD), but we exclude it from our algorithm since it is not always safe.

Given two indexicals \( f_1 \equiv x \text{ in } r_1(y) \) and \( f_2 \equiv y \text{ in } r_2(x) \) which are equivalent. The optimization states that if \( f_1 \) prunes \( x \) in \( \sigma \), thus \( x \) is added to the queue and \( \sigma \) is updated, then when \( x \) is dequeued and \( f_2 \) is considered, in \( \sigma' \) say, \( f_2 \) can be ignored since \( \sigma' \) should be a fixed point of \( f_2 \). This may not be true however.

**Example 6.2:** Suppose we have the following two indexicals:

\[
\begin{align*}
  f_1 &\equiv x \text{ in min}(y) + 1 \cdot \text{max}(y) + 1, \\
  f_2 &\equiv y \text{ in min}(x) - 1 \cdot \text{max}(x) - 1
\end{align*}
\]

which are equivalent. Then consider the store \( \sigma = \{ x \in \{3, 4, 5\}, y \in \{2, 4\} \} \). It follows that \( \sigma \) is a fixed point of the two indexicals. Now suppose \( \sigma \) is updated to \( \sigma_1 = \{ x \in \{3, 4\}, y \in \{2, 4\} \} \), hence, \( f_2 \) is executed, and \( \sigma_1 \) is updated to \( \sigma_2 = \{ x \in \{3, 4\}, y \in \{2\} \} \). However, \( \sigma_2 \) is not a fixed point of \( f_1 \), since \( f_1(\sigma_2) = x \in \{3\} \), which is not entailed by \( \sigma_2 \).
Separate entailment checking from arc-consistency

The following optimization is applicable primarily in a deep-guard concurrent constraint language such as AKL or Oz, where arbitrary combinations of entailment checking expressions, and consistency checking expressions can occur.

Indexicals used for entailment checking should be isolated from those used for arc-consistency, so that indexicals which are not monotone, but used for arc-consistency, can be ignored by the solver until monotone. However, in the following we ignore this detail.

6.3 The Optimized FD Solver

We now give the optimized version of the basic algorithm, using the optimizations in the previous section.

Let $Q$ be a finite queue/set of variables, all stamped with the time of their queueing, $T$ be a (global) time stamp, and let $\sigma$ be a constraint store.

Algorithm 6.3:

```plaintext
variable_check($Q$, $\sigma$) {
    while $Q$ not empty do {
        set $Q = Q \setminus \{y\}$;
        let $F$ be the set of indexicals suspended on $y$;
        for each $f \equiv x$ in $r$ in $F$ do {
            if $f$ marked entailed then dismiss $f$ and continue;
            if $r$ not affected by the pruning of $y$ then continue;
            if (stamp of $f$ $\ge$ stamp of $y$) then continue;
            let $I = x_\sigma \cap r_\sigma$;
            set stamp of $f$ equal to $T$;
            case $I$ of {
                $\emptyset$: if $r$ monotone in $\sigma$ then fail else continue;
                $x_\sigma$: if $r$ antimonotone in $\sigma$ then
                    mark $f$ as entailed and dismiss it;
                    continue;
                otherwise: if $r$ not monotone in $\sigma$ then continue;
                    increment $T$;
                    set stamp of $x$ equal to $T$;
                    set $\sigma = \sigma \cup \{x \in I\}$;
                    if $x$ queued then update type of pruning of $x$;
                    else set $Q = Q \cup \{x\}$;
                    if $r$ constant in $\sigma$ then
```

mark \( f \) as entailed and dismiss it;

} 
}
}
return \( \sigma \);
}

The worst-case complexity of the algorithm is the same as for the basic algorithm, i.e. \( O(nmec) \), where \( n \) is the number of variables, \( m \) the maximum domain size of any variable, \( e \) the maximum number of suspended indexicals of any variable, and \( c \) the cost of evaluating a range. However, its average-case behavior is better (Chapter 10).

\section*{6.4 Data Structures for FD}

There are two basic notions that need to be represented; finite domain \textit{variables} and \textit{indexicals}. Their representations are mutually dependent. A variable must refer to any indexical which contains the variable, and an indexical must refer to the variable it is constraining. Remember that the behavior of an indexical is very much like a reactive process in an operating system, which suspends and wakes depending on the external activity. Hence, pruning a variable will possibly wake the indexicals suspended on the variable, and executing an indexical may possibly prune the variable constrained by the indexical.

\subsection*{6.4.1 The finite domain variable}

There are three important parts of information connected to a finite domain variable. One part concerns the contents of the domain, e.g. the elements. Another part concerns the suspended indexicals (and constraints) linked to the variable. Finally, there is information regarding the optimizations performed by the solver.

More specifically, the domain of a variable can be represented in several ways:

- as a set, preferably by a \textit{bitvector},
- as an \textit{interval} when possible, i.e. by two numbers, or
- by a mixed representation of sets and intervals.

In practice, the domain is usually initialized as an interval and converted to a set when elements inside the interval are removed by the solver. Our experience is that the mixed representation is cumbersome and hardly worthwhile.

By restricting all finite domain variables to be contained in the natural numbers, the representation of bitvectors is simplified considerably, since otherwise the bitvectors must be adjusted and normalized throughout the
computations. This does not limit the expressiveness of the indexicals however, and therefore we assume from now on that each domain variable is bounded by a set of natural numbers.

Furthermore, information should be kept regarding the current maximum and minimum values of the domain, as well as of the size, even though not strictly necessary.

For each domain variable $x$, all indexicals referring to the domain of $x$ must themselves be referred to by $x$ so that whenever the domain of $x$ is pruned, the indexicals can be reconsidered.

Finally, some information is needed to implement the optimizations exploited by the solver (Section 6.2).

Thus to summarize; the representation of a finite domain variable should contain:

- the current minimum and maximum values of the variable,
- if the variable is not an interval, a bitvector containing the elements of the domain, and thus the size of the domain,
- the time and type of the latest pruning of the variable,
- a link field used for queueing the variable when pruned,
- a list of suspended indexicals. In fact, several lists can be used; one for all indexicals which depend on DOM prunings of the variable, one for all indexicals which depend on MIN prunings of the variable, one for all which depend on MAX prunings, and one for all which depend on the variable being determined (INT dependencies) [DC93b].

### 6.4.2 The indexical

The representation of the indexical should contain enough information to allow the indexical to be evaluated in any store, and also guarantee that the optimizations in Section 6.2 can be implemented.

Consider the indexical $x$ in $r$. Its representation must contain:

- A reference to $x$.
- A reference to $r$, or rather, a reference to the representation of $r$ (Section 6.6).
- The arguments of $r$, i.e. the environment (call) frame of $r$, containing all variables which occur in $r$.
- An entailment flag connecting the indexical to other, equivalent, indexicals.
- The time stamp for when the indexical was last executed.
Decision information used for classifying the monotonicity of $r$ at run-time (Algorithm 6.5).

In Section 9.1 the particular representation of indexicals in AKL(FD) is given. See also [DC93c, DC93b].

### 6.5 The Monotonicity Algorithm

Assume we are given the indexical $x$ in $r$, of which the representation contains two sets $M_r$ and $A_r$, initially set to $M_r$ and $A_r$ (Section 3.5) respectively.

The algorithm for computing the monotonicity of $r$ in a store $\sigma$ is thus defined as:

**Algorithm 6.5:**

```plaintext
monotonicity($M_r$, $A_r$, $\sigma$) {
    set $M_\sigma = M_r \setminus \{x: x$ determined in $\sigma\}$;
    set $A_\sigma = A_r \setminus \{x: x$ determined in $\sigma\}$;
    if $M_\sigma = A_\sigma = \emptyset$ then return CONSTANT;
    if $M_\sigma = \emptyset$ then return MONOTONE;
    if $A_\sigma = \emptyset$ then return ANTIMONOTONE;
}
```

Everytime the indexical is evaluated in a store $\sigma$, $M_r$ and $A_r$ are replaced by $M_\sigma$ and $A_\sigma$ respectively. Hence, once all the variables in $M_r$ ($A_r$) are determined, in store $\sigma$ say, the monotonicity test will be immediate in all extensions of $\sigma$.

### 6.6 The FD Emulator

The FD emulator evaluates ranges in a store $\sigma$, i.e. given a range $r$ it computes $r_\sigma$. We use a simple-minded abstract machine, where the emulator is stack-based. Each FD operator is interpreted by a corresponding instruction, which takes its arguments from the stack. We have added a number of special instructions to optimize common cases.

The emulator is thus provided an index to byte code and the call frame $a$ for the code, represented by a vector of (domain) variables. Furthermore, the emulator operates on three data areas: the code area $c$, the value stack $v$, containing integers or references to domains, and the domain heap $d$, containing all intervals and sets constructed in the emulation. At each invocation of the emulator, $d$ is assumed empty, where there can be no references into $d$ from other areas of the memory. Hence, all memory in $d$ is automatically reclaimed at each invocation.
The instructions are straightforward and defined in Table 6.2, where \( c_i \) is the current instruction dispatched on, \( d \) and \( d' \) denote domains, \( m \) and \( n \) denote integers, and the arithmetic operators are applied pointwise where appropriate.

### 6.6.1 The emulate kernel

Let \( i \) be the current code index, and \( j \) the current top of the value stack. The instructions are straightforward and defined in Table 6.2, where \( c_i \) is the current instruction dispatched on, \( d \) and \( d' \) denote domains, \( m \) and \( n \) denote integers, and the arithmetic operators are applied pointwise where appropriate.

### 6.6.2 Optimized instructions

Special instructions can be added to the emulator for optimizing the evaluation of indexicals used for checking arc-consistency of common arithmetic (Section 9.3) and symbolic constraints. We give some examples in Table 6.3. The basic idea is simple: small instructions are combined into larger, thereby decreasing the emulation time. However, the converse optimization is equally valid, if compilation to RISC-based machine code is considered.
### Table 6.3: Optimized byte code instructions

<table>
<thead>
<tr>
<th>$c_i$</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{div_c}$</td>
<td>$v[j++] = \lfloor \min(c_{i+1})/c_{i+2} \rfloor \cdots \lfloor \max(c_{i+1})/c_{i+2} \rfloor$; $i = i + 3$;</td>
</tr>
<tr>
<td>$x_{mult_c}$</td>
<td>$v[j++] = \min(c_{i+1}) \cdot c_{i+2} \cdot \max(c_{i+1}) \cdot c_{i+2}$; $i = i + 3$;</td>
</tr>
<tr>
<td>$x_{add_c}$</td>
<td>$v[j++] = \min(c_{i+1}) + c_{i+2} \cdot \max(c_{i+1}) + c_{i+2}$; $i = i + 3$;</td>
</tr>
<tr>
<td>$x_{add_c_min}$</td>
<td>$v[j++] = \min(c_{i+1}) + \min(c_{i+2}) \cdot \max(c_{i+1}) + \max(c_{i+2})$; $i = i + 3$;</td>
</tr>
<tr>
<td>$x_{add_c_max}$</td>
<td>$v[j++] = 0 \cdot \max(c_{i+1}) + c_{i+2} \cdot \max(c_{i+1}) + \max(c_{i+2})$; $i = i + 3$;</td>
</tr>
<tr>
<td>$x_{add_y}$</td>
<td>$v[j++] = \min(c_{i+1}) + \min(c_{i+2}) \cdot \max(c_{i+1}) + \max(c_{i+2})$; $i = i + 3$;</td>
</tr>
<tr>
<td>$x_{sub_y}$</td>
<td>$v[j++] = \min(c_{i+1}) - \max(c_{i+2}) \cdot \max(c_{i+1}) - \min(c_{i+2})$; $i = i + 3$;</td>
</tr>
<tr>
<td>$x_{mult_y_mult_c}$</td>
<td>$v[j++] = \min(c_{i+1}) \cdot \min(c_{i+2}) \cdot c_{i+3}$; \max(c_{i+1}) \cdot \max(c_{i+2}) \cdot c_{i+3}$; $i = i + 4$;</td>
</tr>
<tr>
<td>$x_{div_y_mult_c}$</td>
<td>$v[j++] = \lfloor \min(c_{i+1})/\min(c_{i+2}) \cdot c_{i+3} \rfloor \cdots \lfloor \max(c_{i+1})/\max(c_{i+2}) \cdot c_{i+3} \rfloor$; $i = i + 4$;</td>
</tr>
<tr>
<td>$element_x$</td>
<td>$l = c_{i+2}; v[j++] = \text{dom}{l_k:k \in \text{dom}(c_{i+1})}$; $i = i + 3$;</td>
</tr>
<tr>
<td>$element_i$</td>
<td>$l = c_{i+2}; v[j++] = \text{dom}{k:l_k \in \text{dom}(c_{i+1})}$; $i = i + 3$;</td>
</tr>
</tbody>
</table>

thus instead generating fine-grained instructions.

### 6.7 The FD Compiler

The FD compiler translates ranges into byte code as described in Section 6.6, computes the dependency information needed to implement the optimization in Section 6.2, and extracts the sets used by the monotonicity algorithm (Algorithm 6.5).

#### 6.7.1 Translation into byte code

The translation of a range $r$ into byte code is straightforward, and is defined recursively by Table 6.4 and Table 6.5. Basically, a postfix Polish notation is used, where we assume an argument mapping $A_x$ is given, such that $A_x$
equal the position of $x$ in the argument frame. The byte code for a range $r$ is always terminated with a halt instruction.

### 6.7.2 Additional processing

The compiler must also generate dependency and monotonicity information, and a flag used for the entailment marking.

The monotonicity information for a range $r$ is computed by Table 3.4 extended with rules for the symbolic functions (Table 6.6).

Furthermore, the dependencies for all domain variables in $r$ are computed as in Section 6.2 with one addition. If the range occurs in an indexical used for arc-consistency, the dependency for each variable $x$ in $M_r$ is set to INT, and if the indexical is used for entailment checking, the dependency of each variable $x$ in $A_r$ is set to INT. This is an optimization which reduces the number of calls to the monotonicity algorithm (Chapter 10).
The compiler can take advantage of the special instructions added to the emulator (Section 6.6.2) as in Table 6.7, where ranges used for defining library calls (see Section 9.3) could be peephole optimized. The instructions \texttt{element \_i}(x, [l_1, \ldots, l_k]) and \texttt{element \_i}(i, [l_1, \ldots, l_k]) together define the element \texttt{(i, [li_1, \ldots, li_k])} constraint as in Appendix A. Also, see Appendix A for definitions of the library constraints for arithmetics, which could exploit the peephole optimizations. However, our current implementation does not use these optimizations.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
$r$ & $T_r$ & $T_v$ \\
\hline
$\text{min}(x) / c \ldots \text{max}(x) / c$ & $x \_\text{div} \_c \ x \_a \ x \_c$ & $x \_\text{div} \_c \ x \_a \ x \_c$ \\
$c * \text{min}(x) / c \ldots \text{max}(x) / c$ & $x \_\text{mult} \_c \ x \_a \ x \_c$ & $x \_\text{mult} \_c \ x \_a \ x \_c$ \\
$\text{min}(x) + \text{min}(y) / c \ldots \text{max}(x) + \text{max}(y) / c$ & $x \_\text{sub} \_y \ x \_a \ x \_y$ & $x \_\text{add} \_y \ x \_a \ x \_y$ \\
$c * \text{min}(x) / c \ldots \text{max}(y) / c$ & $x \_\text{add} \_c \ x \_a \ x \_c$ & $x \_\text{add} \_c \ x \_a \ x \_c$ \\
$\text{min}(x) + c \_a \ldots \text{max}(x) + c \_a$ & $x \_\text{add} \_c \ x \_a \ x \_c$ & $x \_\text{add} \_c \ x \_a \ x \_c$ \\
$\text{min}(x) + c \_a \ldots \text{max}(y) / c$ & $x \_\text{mult} \_y \ x \_a \ x \_c$ & $x \_\text{mult} \_y \ x \_a \ x \_c$ \\
$\text{min}(x) / c \ldots \text{max}(y) / c$ & $x \_\text{div} \_y \ x \_a \ x \_y$ & $x \_\text{div} \_y \ x \_a \ x \_y$ \\
\hline
\end{tabular}
\caption{Peephole translation of ranges to byte code}
\end{table}

Furthermore, the fact that each code sequence is terminated by \texttt{halt} can be used for optimizing the emulator. Instead of letting the emulator compute a domain which then is intersected with a variable \texttt{x} (Section 6.3), the \texttt{halt} instruction can be combined with the previous instruction to do the intersection without generating an intermediate domain. Thereby, of course, taking necessary precautions regarding destructive assignments.

Finally, indexicals that are annotated as equivalent are provided a common entailment flag used for the optimization in Section 6.2.

### 6.7.3 Compilation for arc-consistency

We now give examples of ranges compiled for arc-consistency.

**Example 6.7.3:** Consider

\[ r \equiv \text{dom}(x) \setminus (\text{dom}(y) + n). \]
It follows that $\mathcal{M}_r = \{y, n\}$. Therefore, the dependencies are $y/\text{INT}$ and $n/\text{INT}$.

Assume $A_x = 0$, $A_y = 1$, and $A_n = 2$. The byte code for $r$ is thus

```
val 2
dom 1
set_add
dom 0
diff
halt
```

Now, an example of how the special arithmetic instructions can be exploited.

**Example 6.7.3:** The range $\min(x)+\min(y)$. $\max(x)+\max(y)$ can either be compiled as, assuming $A_x = 0$ and $A_y = 1$:

```
max 1
max 0
add
min 1
or as, recognizing a special case: $x$. $\text{add}$. $y$. 0 1
min 0
add
range
halt
```

---

**6.7.4 Compilation for entailment checking**

We now give an example of how a range is compiled for entailment. Consider

$$r \equiv 0.(\min(x) - \max(y)).$$

It follows that $\mathcal{A}_r = \emptyset$. Therefore, the dependencies are: $x/\text{MIN}$ and $y/\text{MAX}$.

Assume $A_x = 0$ and $A_y = 1$. The byte code for $r$ is thus

```
dom_max 1
dom_min 0
sub
const 0
range
halt
```

**6.7.5 Other optimizations**

It should be clear that many traditional optimizations apply to the FD emulator and compiler [ASU86]. These include identifying common subexpressions, using a register-based machine and exploiting the registers for storing
values frequently referred to, and perhaps compiling to low-level machine code, thereby, adapting to the underlying architecture. For example, as have been mentioned, a RISC-oriented instruction set is an interesting alternative if machine-code compilation is to be used. However, we do not further address these issues here.
Chapter 7

AKL

In this chapter we describe AKL, and the constraint store and guard model of AKL [Jan94]. Our contribution to the design and implementation of AKL has been the constraint interface and constraint lifting [CJM93, CC95].

7.1 Introduction

Concurrent constraint programming (CCP) is a powerful paradigm for programming with constraints, based on simple concepts [Mah87, Sar89]. A set (or conjunction) of constraints, regarded as formulas in first-order logic, forms a constraint store. A number of agents interact with the store using the two operations tell, which adds a constraint to the store, and ask, which tests if the store either entails or disentails the asked constraint, otherwise waiting until it does. Telling and asking correspond to sending and receiving “messages”, thereby providing the basic means for communication and synchronization for concurrent programming.

The Agents Kernel Language (AKL) is a concurrent constraint programming language which generalizes the above functionality using a small set of powerful combinators [JH91, Jan94]. The basic paradigm is still that of agents communicating over a constraint store, but the combinators make possible also other readings, depending on the context, where agents compute functions or relations, serve as user-defined constraints, or as objects in object-oriented programs. A major point of AKL is that its paradigms can be combined. For example, it is quite natural to have a reactive process- or object-oriented top-level in a program, with other components performing constraint solving using don’t know nondeterminism. The nondeterminism can be encapsulated, so that it does not affect the process component.

Nondeterminism in AKL is controlled using stability, a generalization of the Andorra principle, which has proven its usefulness in the context of constraint programming (see e.g., [GY92]). Basically, a computation
configuration is stable if no external activity can influence the outcome of the configuration.

The notion of constraints in AKL is generic, however, as a basic builtin constraint system there are the rational trees. A constraint system in AKL must provide algorithms for entailment and consistency, neither of which need to be complete. Consistency is maintained through constraint propagation and checking.

A programming environment for AKL is currently being developed at SICS [JM92]. Among the goals are to: (1) develop the necessary implementation technology for efficient (sequential and parallel) execution, (2) develop an execution model which allows different constraint systems to be easily incorporated, (3) offer good interoperability with conventional languages such as C, and (4) investigate large scale applications where combinations of paradigms naturally occur.

7.2 AKL: the Language

This section gives a brief and informal summary of the kernel of AKL, a deep-guard, don’t know nondeterministic CCP language. However, we exclude ports and aggregates. For a complete definition see [JH91], [Fra94], or [Jan94].

7.2.1 The language

Basic statements, such as constraints, calls, composition, and hiding, are as in other CCP languages [Sar93] (see Figure 7.1). A constraint \( c(\overline{x}) \) executes by calling the appropriate constraint solver, which simplifies the constraint and checks its consistency and/or entailment with the current store. A call \( p(\overline{x}) \) is replaced by its definition, and a composition \( S_1; S_2 \) executes \( S_1 \) and \( S_2 \) concurrently. The hiding statement \( \overline{x} : S \) introduces local variables, which are invisible to the surrounding environment.

\[
S ::= c(\overline{x}) \quad \text{constraint} \\
    | p(\overline{x}) \quad \text{call} \\
    | S; S \quad \text{composition} \\
    | \overline{x} : S \quad \text{hiding} \\
    | (C_1; \cdots; C_k) \quad \text{choice} \\
C_k ::= \overline{x} : S \% S \quad \text{clause} (\% \in \{\rightarrow, ?, !\}) \\
D ::= p(\overline{x}) := S. \quad \text{definition}
\]

Figure 7.1: Syntax of statements in AKL
The particular flavor of AKL is given by the deep-guard choice statements. Choice statements are sequences of clauses. The first statement of a clause is the guard and the second the body. The guard operator % can be either of →, ?, or [, and corresponds to conditional choice, nondeterminate choice, or committed choice statements, respectively.

The guards of choice statements execute with corresponding local stores. Since arbitrary statements are allowed in guards, the stores in an execution state form a hierarchy. If the union of a local store with the external stores is inconsistent, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, a choice statement fails.

The conditional choice corresponds to if-then-else in Prolog, where entailment is used for checking the guard constraints. If the first remaining guard is reduced to a store that is entailed by the union of external stores, the statement is replaced with the body of this clause and the constraints in its store are moved to the closest external store. The clause is thus said to be promoted.

Nondeterminate choice corresponds to disjunction in Prolog. If only one clause remains, and its guard is successfully reduced to a store which is consistent with the union of external stores, the choice statement is said to be determinate. The clause is then promoted.

Otherwise, if there is more than one clause left, the choice statement is said to be nondeterminate, and it will wait (suspend). Subsequent telling of other agents may make it determinate. If eventually a state is reached in which no other computation step is possible, even by subsequent telling in external stores, each of the remaining clauses may be promoted in different copies of the state. The alternative computation paths are explored concurrently. The state reached is called stable, and the behavior is a generalization of the Basic Andorra Model to deep guards.

Up to this point, the constructs introduced belong to the strictly logical subset of AKL, which has an interpretation in first-order logic both in terms of success and failure [Fra94].

Committed choice corresponds to guarded clauses in committed-choice languages. If any of the guards is successfully reduced to a store which is entailed by the union of external stores, the corresponding clause is promoted.

In Section 8.5 we will see how the choice statements of AKL can be used for constraint programming. A clausal syntax for definitions with choice statements will be used, which corresponds to the above syntax in a straightforward manner.

Also, we want to point to a recent addition to AKL, the solve-combinator solve(A,Z), where A is a lambda-abstraction, and Z is computed to the result of evaluating A, where Z can equal failed meaning that A fails when evaluated,
solved(S) where is an abstraction which when evaluated returns the only solution to A, and
distributed(L,R) where L and R are two abstractions, containing the leftmost alternative to A in L, and the rest of the alternatives in R.

Thereby, solve/2 replicates the solve-combinator of Oz [SS94]. Henceforth, branch-and-bound techniques and similar can be programmed at source-level. For example,

\[
\text{less_abs(A)} : - \\
A = (X,Y)(X < Y). \quad \% \equiv \lambda x, y. x < y
\]

\[
\text{greater_abs(A)} : - \\
A = (X,Y)(X > Y).
\]

\[
\text{bab(L, Order, Best, X)} : - \\
L = [] \\
\rightarrow X = \text{Best}.
\]

\[
\text{bab(L, Order, Best, X)} : - \\
L = [G|L1] \\
\rightarrow \text{solve}(Z \setminus (\text{Order}(Z, \text{Best}), G(Z)), R), \\
\text{bab_solution}(R, L1, Order, \text{Best}, X).
\]

\[
\text{bab_solution}(R, L, Order, X) : - \quad \% \text{new optimum} \\
R = \text{solved}(S) \\
\rightarrow S(B1), \\
\text{bab}(L, \text{Order}, B1, X).
\]

\[
\text{bab_solution}(R, L, Order, B, X) : - \quad \% \text{old optimum remains} \\
R = \text{failed} \\
\rightarrow \text{bab}(L, Order, B, X).
\]

\[
\text{bab_solution}(R, L, Order, B, X) : - \quad \% \text{several remaining choices} \\
R = \text{distributed}(\text{Left}, \text{Right}) \\
\rightarrow \text{bab}(\text{Left}, \text{Right}|L1, \text{Order}, B, X).
\]

initially called as \text{bab}([A], \text{Order}, \text{Init}, X) for some goal A and ordering abstraction (e.g. less_abs(Order)), where \text{Init} is large number or small number depending on Order. Thereby, the smallest (largest) solution to A is computed using branch-and-bound.

We conclude the presentation of AKL with an example.

7.2.2 An example

The following is a program which performs cardinality-like reasoning. Given a list of elements, \text{count}(N,L,M) asserts that the number of elements in L, which are equal to M, equals N. This is done by associating a boolean variable (as in Section 3.6.7) with each equality, and checking that the sum of the boolean variables equals N. The use of committed-choice in \text{merge/3} is used to collect the boolean values in an arbitrary order.
count(N,L,M) :-
  constrain_list(L, M, S),
  sum_list(S, 0, N).

constrain_list([], _, S) :-
  S=[].
constrain_list([X|R], M, S) :-
  check(X, M, S0),
  constrain_list(R, M, S1),
  merge(S0, S1, S).

check(X, X, S) :-
  S=[1].
check(_, S) :-
  S=[0].

sum_list([], N, M) :-
  N=M.
sum_list([B|R], N, M) :-
  N<M
  ->
  N1 is B+N,
  sum_list(R, N1, M).
sum_list([B|R], N, N) :-
  ->
  zero_list([B|R]).

zero_list([]) :-
  ->
  true.
zero_list([0|R]) :-
  ->
  zero_list(R).

merge([], Y, Z) :-
  Y=Z.
merge(X, [], Z) :-
  X=Z.
merge([A|X], Y, Z) :-
  Z=[A|Z0],
  merge(X, Y, Z0).
merge(X, [A|Y], Z) :-
  Z=[A|Z0],
  merge(X, Y, Z0).

Furthermore, label(L, I, J) enumerates solutions to L which can thus be
checked by count/3 for a given N and M. The predicate between/3 is non-
deterministic by the use of the wait-choice.

label([], _) :-
  ->
  true.
label([X|R], N, M) :-
\[
\rightarrow \text{between}(X, N, M), \\
\quad \text{label}(R, N, M).
\]
\[
\text{between}(X, N, M) :- \\
\quad X=N \\
\quad \text{? true}.
\]
\[
\text{between}(X, N, M) :- \\
\quad N<M \\
\quad \text{? N1 is N+1,} \\
\quad \text{between}(X, N1, M).
\]

Consider now the goal \(\text{count}(2,[X1,X2,X3],4),\text{label}([X1,X2,X3],4,5)\). The call \(\text{count}(2,[X1,X2,X3],4)\) generates three suspended calls \(\text{check}(X1,4,S1)\), \(\text{check}(X2,4,S2)\), and \(\text{check}(X3,4,S3)\), together with the corresponding calls to \(\text{merge}(S1,S4,S)\), \(\text{merge}(S2,S3,S4)\), and to \(\text{sum_list}(S,0,2)\).

Then \(\text{label}([X1,X2,X3],4,5)\) nondeterministically determines \(X1\) to 4, and \(S1\) is thereby bound to \([1]\), and hence \(S\) is bound to \([1|S]\). Next, the call to \(\text{label}([X2,X3],4,5)\) nondeterministically determines \(X2\) to 4, and \(S2\) is bound to \([1]\), and hence \(S'\) is bound to \([1|S']\). Now, the limit 2 is reached, and \(\text{zero_list}(S'')\) constrains \(S''\) to contain only 0. Thus, when \(\text{label}([X3],4,5)\) tries \(X3=4\), \(S'\) is bound to \([1]\) and \(\text{zero_list}(S'')\) fails. Instead \(X3\) is set to 5. The first solution computed will thus be \(X1=4\), \(X2=4\), and \(X3=5\).

### 7.3 Constraint stores in AKL

A constraint system for AKL may in principle be any first-order theory, closed under conjunction and existential quantification, for which decision procedures for satisfiability and entailment can be provided. Of course, a large body of work on practically motivated constraint systems exists [JM94]. We have so far considered constraint systems over rational trees, feature trees, and finite domains in AGENTS [Kei94, Jan94, CJH94]. This investigation will be extended to other systems in the future, such as systems over boolean and rational numbers.

The rational tree constraint system is the same as in SICStus Prolog and Prolog-II [Col84], and has been inlined in the AKL-emulator for efficiency reasons. Our performance evaluations indicate that AKL is as fast as Prolog in doing constraint solving over such equalities [JM92].

We now describe the constraint store model, and the generic constraint protocol, of AKL.
7.3.1 Constraint stores

Seen from a logical perspective, stores in an AKL computation state are of the following general form.

\[ P ::= \exists V (\theta \land Q \land \cdots \land Q) \]

where each \( Q \) is of the form

\[ Q ::= (P \lor \cdots \lor P) \]

For simplicity, let us call components of the \( P \) form \textit{conjunctions} and expressions of the \( Q \) form \textit{disjunctions}. Conjunctions contain \textit{constraint stores} \( \theta \). Variables in \( V \) are said to be \textit{local} in a conjunction.

Computations are performed by a worker moving about in the computation state, represented by a tree, performing rewrites corresponding to the different possible transitions between states. The \textit{environment} of a rewritten component is the set of stores \( \theta \) in the conjunctions “surrounding” it. When moving in the computation state, the worker maintains an efficient representation of the current environment.

Non-determinism in AKL is obtained by splitting part of a computation state in two branches. This is done simply by replacing

\[ \exists V (\theta \land (A \lor B) \land Q) \]

with

\[ \exists V (\theta \land A \land Q) \lor \exists V (\theta \land B \land Q) \]

and applying suitable simplifications. The duplicated parts may be rewritten quite differently. Such differences can be maintained by trailing, but, for simplicity, the AGENTS implementation performs copying corresponding to the duplication of components in the above schematic rule.

In AKL, disjunctions are created by the execution of the various choice statements, and the conjunctions they contain correspond to the execution of guards. Disjunctions and conjunctions also occur at the top-level and in aggregates.

A guard computation in AKL is hence embedded within a private local store, and no distinction is made between execution inside or outside guards. This fundamental design choice enables encapsulated (non-determinate) computations as well as logically more complete implementations. The latter follows simply because guards involving inconsistent and mutually incompatible constraints can be discarded.

In the design of AGENTS we have adopted a view of constraints mimicking the constraint view of Prolog, i.e., we represent the constraints through the variables they constrain. Thus, from the implementation point of view, we consider the store not to be a set of constraints [SRP91], but a set of variables. Any constraint \( c(X) \) adds its information on \( X \), ensuring that \( X \) contains enough information to recover the meaning of \( c \).
In the following we use C syntax in describing the outline of the structures involved. A constraint variable is thus layed out as:

```c
struct {
    varMethod *method;
    id *env;
    suspension *susp;
}
```

where `method` is a vector of functions that may be applied to a variable, `env` is a tag identifying the locality of the variable, and `susp` is a list of predicate calls that are waiting for the variable to receive a value.

The function vector contains the following functions:

```c
varMethod method =
{
    new,
    unify,
    print,
    copy,
    gc,
    copy_external,
    gc_external
}
```

Except for `copy_external` and `gc_external`, the purpose of the functions should be obvious. `copy_external` (as `gc_external`) is called by the copying (garbage collecting) procedure to allow a constraint variable to duplicate any suspensions of copied goals.

### 7.3.2 Constraint operations

The (sequential) AGENTS worker is basically a transformer of configurations, where a configuration consists of a constraint store hierarchy, as above, and the occurrence of the interpreter. Each conjunction `P` refers to a list of suspended constraints `PC`. The environment of `P` is referred to as `Pb`. The worker rewrites the configurations according to the operational semantics of AKL [JH91, Jan94].

Suppose a constraint `c` is executed in `P`. If `c` and `Pb` are inconsistent, the execution fails and `P` is marked as dead. If `c` is entailed, the computation succeeds. Otherwise, `c` is simplified with respect to `Pb`, and added to `PC`. Thus, the execution of `c` terminates.

A constraint `c` is suspended in `P` by adding `c` to `PC`. This way, the worker is provided an object by which the management of the store hierarchy is possible (see below).

The worker exits a conjunction `P` when `PC` is not entailed by `Pb`, and the associated guard operator, such as `\|`, of `P` expects entailment. Hence,
$P$ is *deinstalled*, i.e. $P_C$ is retracted from $P_0$. This requires that the representation of $P_C$ is kept explicit.

The worker *installs* a conjunction $P$ when some update to $P_0$ has affected the value of variables constrained in $P_C$. Hence, if $P_C$ is inconsistent with $P_0$, the installation fails, if $P_C$ is entailed by $P_0$ the installation succeeds, and otherwise the installation suspends and the conjunction is deinstalled.

Suppose $P$ occurs in

$$Q = (P_1 \lor \cdots \lor P \lor \cdots \lor P_k),$$

where $Q$ occurs in a conjunction

$$P' = \exists W (\gamma \land Q_1 \land \cdots \land Q \land \cdots \land Q_l).$$

If $Q$ is replaced with $P$ in $P'$, e.g. by committing to a certain guard, $P$ is *promoted*, i.e. $P'$ becomes

$$\exists W \lor V (\gamma \lor \theta \land Q_1 \land \cdots \land P \land \cdots \land Q_l)$$

and $P_C$ is added to $P'_C$. Variables previously external in $P_C$ may now be local, since $W$ and $V$ have been merged, and thus constraints in $P_C$ may become entailed by the promotion. The internal representation of the constraints can be optimized by the promotion, since information that was needed for deinstallation is now redundant and can thus be removed. Furthermore, $P$ is marked as dead and linked to $P'$ such that any object local in $P$ becomes local in $P'$.

On the low level a constraint is represented as:

```c
struct constraint {
    constraintMethod *method;
    struct constraint *next;
}
```

The *next* field is used to link constraints into the list of locally declared constraints. The *method* field is a function vector which contains the following functions:

```c
constraintMethod method = {
    install,
    reinstall,
    deinstall,
    promote,
    print,
    copy,
    gc
}
```

where the meaning of the functions should be obvious, except for the function *reinstall* which is called after copying or garbage collecting parts of the memory. *Reinstallation* is a special case of installation where the installed constraints cannot fail nor propagate.
7.4 Conclusion

Conceptually, a concurrent constraint programming (CCP) language is an ideal vehicle for constraint programming. The expressiveness of the constraint solver can be extended by user-defined entailment-driven propagation rules that execute concurrently and cooperate with the constraint solver. However, to offer an orthogonal combination of constraint programming with the other paradigms offered by CCP—concurrent, relational, functional, object-oriented, ...—the language must be deep. Being deep means having a hierarchy of constraint stores, where a computation need not be affected by the failure of a subordinate store. This makes it possible to have a reactive process/object-oriented top-level in a program, with other encapsulated components performing constraint solving.

AKL is a deep CCP language [JH91, Jan94]. This enables, for example, a simple and complete implementation of constraint lifting which can be used to execute disjunctions of finite domain constraints (Section 8.3 and 9). We will now move on to AKL(FD), the extension of AKL with FD, and consider in more detail its design, use (Chapter 8), implementation (Chapter 9), and efficiency (Chapter 10).
Chapter 8

AKL(FD)

The design of AKL(FD) is given, together with several examples of programming in AKL(FD). This chapter is based on [CJH94, CC93]. The implementation details are given in Chapter 9, and a performance evaluation of the system is given in Chapter 10.

8.1 Introduction

The AGENTS implementation of AKL [JMB+94], being developed at SICS, has rational trees as its basic constraint system, which is supported efficiently at the emulator level. Other constraint systems, such as FD, are integrated using generic variables and generic constraints, which are objects offering methods for the services required by the emulator, such as variable binding, garbage collection, and global propagation. This enables a simple integration of arbitrary constraint systems with reasonable efficiency.

In this chapter we illustrate by examples how nontrivial arithmetic and symbolic finite domain constraints can be defined as AKL(FD) programs, constraints that have to be provided as primitives in previous CLP systems. Some of the constraints use conditional and disjunctive reasoning, which is formulated completely within AKL(FD). We also give an example of how cardinality reasoning is captured in AKL(FD). Monotone FD indexicals are used as programming primitives when defining finite domain constraints used for arc-consistency checking.

Furthermore, we show that both consistency and entailment checking versions of the constraints can be defined, again within AKL(FD). The entailment of finite domain constraints are thus checked by AKL(FD) programs, using antimonotone FD rules as primitives to implement entailment checking. We also stress the use of monotonicity reasoning, when programming with FD, to control the suspension of constraints.
In AKL(FD), a programmer may use indexicals directly, and has also the option to rely on the AKL(FD) constraint compiler to produce appropriate indexicals for given constraints (Section 9.4). For a sequence of linear arithmetic constraints \( c \), a statement \( \text{fd}(c) \) is compiled to monotone indexicals, suitable for consistency checking, a statement \( \text{fd}_{\text{ask}}(c) \) is compiled to an antimonotone indexical, suitable for entailment checking, and a statement \( \text{fd}_{\text{or}}(c) \) is compiled such that the disjunction is applied constructively.

Furthermore, builtins are provided for enumerating solutions. The builtin \( \text{enum}(M,L) \) enumerates solutions to the variables in \( L \), using the method \( M \), where \( M \) currently can be naïve or first-fail enumeration, either using a copying-based, or a trailing-based search, or constructive enumeration, using a copying-based search. For an explanation of the different search principles see Section 9.5. Finally, a recent addition of a solve-primitive to AKL has enabled branch-and-bound techniques to be programmed at source-level (Section 7.2).

Also, we have included a number of common symbolic constraints such as \( \text{atleast} / 3 \), \( \text{atmost} / 3 \), \( \text{count} / 3 \), and \( \text{element} / 3 \) (Section 3.6.7).

User-defined finite domain constraints are simply defined as AKL(FD)-predicates. We list the basic syntax of AKL(FD) in Table 8.1, where \( N \) and \( R \) are defined in Table 3.1.

In AGENTS, indexicals are compiled to byte code for a simple stack machine that evaluates range expressions (Section 6.6). If during execution it is established (Section 6.3) that an indexical is entailed, it is dismissed.
8.3 Constraint lifting in AKL(FD)

Constraint lifting is disjunction applied constructively to a set of constraint stores generated by running a disjunction of goals in separate local stores, where constraints implied in each local store are lifted and added to the embedding store. The notion is generic in choice of constraint system, and computing approximations of disjunctions generally requires domain specific knowledge. However, for some constraint systems, such as boolean equalities, where the constraint language supports disjunction, lifting becomes trivial.

In AKL, we have introduced constraint lifting through a deep guard operator \( \| \), explained below. Any computation in AKL is done in a local constraint store. A hierarchy of stores is created by running goals in guards. Each local store is associated with all the constraints generated by the local execution, that constrain or depend on variables in external stores. Thus, AKL supports directly the structures that are necessary to implement constraint lifting, since a representation is kept which gives access both to constraint stores and to the constraints visible externally in each store.

The operator \( \| \) is defined as an adaption of the guard mechanism of AKL. Thus, the statement

\[
G_1 \| B_1 \\
\vdots \\
\cdots \\
\vdots \\
G_n \| B_n
\]

in AKL is used for expressing constraint lifting. Its components are called \( \textit{guarded clauses} \) and the components of a clause \( \textit{guard} \ (G_i) \) and \( \textit{body} \ (B_i) \), where \( G_i \) and \( B_i \) contain procedure calls which include constraints.

Now, in the following let \( \alpha \) be a function such that, given the constraint stores \( \sigma_1, \ldots, \sigma_k \), \( \alpha(\sigma_1, \ldots, \sigma_k) \subseteq \sigma_i \) holds, for any \( i \) between 1 and \( k \). For an instance of \( \alpha \) see Section 9.6.

Suppose a lifting statement is executed in a store \( \sigma \). Its guards are executed separately and the execution of the statement proceeds as follows.
Let $\sigma_i$ be the store resulting from the execution of guard $G_i$ in $\sigma$.

- If $\sigma_i$ is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the lifting statement fails.

- If only one nonfailed clause remains, say $\sigma_i$ is promoted, and the lifting statement is replaced with the body $B_i$.

- If $\sigma_i$ is entailed by $\sigma$, the lifting statement is replaced with $B_i$.

- Otherwise, let $\sigma_1, \ldots, \sigma_k$ be all remaining local stores which are neither unsatisfiable nor entailed by $\sigma$, $k > 1$. Hence, if $\alpha(\sigma_1, \ldots, \sigma_k) \not\subseteq \alpha$, add the constraints in $\alpha(\sigma_1, \ldots, \sigma_k)$ to $\sigma$, which invokes the appropriate constraint solver.

- Finally, the lifting statement suspends until more constraints are added to $\sigma$ which may affect the execution of $G_i$, for some $i$, $1 \leq i \leq k$, and thereby the statement is reexecuted (incrementally, of course).

Using the lifting statement, disjunctions of finite domain constraints can be applied constructively through a lifting function $\alpha$ for domain constraints (next section), and by encoding a disjunction $c_1 \lor \cdots \lor c_n$ as $c_1 \true; \cdots; c_n \true$.

### 8.4 FDC in AKL(FD)

Using the guard operators of AKL, combined with the primitive operators of AKL(FD), FDC can thus be coded in AKL(FD) as:

**Domains:** Domain constraints $x \in \{n_1, \ldots, n_k\}$ are coded by $\text{fd}(x \text{ in } n_1 \lor \cdots \lor n_k)$.

**Arithmetic:** An arithmetic constraint $c$ used for arc-consistency is simply written as $\text{fd}(c)$, and if used for entailment checking as $\text{fd\_ask}(c)$.

**Conjunction:** Conjunction of expressions is coded by the composition combinator “,” of AKL.

**Disjunction:** Disjunctions are defined using either the waiting guard “?” or the lifting guard “!”. Furthermore, the $\text{fd\_or}$ operator can be used for expressing disjunctions of constraints, or the disjunction can be programmed using indexicals directly, e.g. exploiting the indexical scheme in Chapter 4.

**Blocking:** Blocking implication is defined using either the conditional guard “→” or the committed-choice guard “[“”, where constraints in the guard can either be wrapped by $\text{fd}$ or by $\text{fd\_ask}$, depending on whether arc-consistency or entailment checking should be used.
Symbolic: Symbolic constraints can be programmed using the above primitives, together with stating indexicals directly through the fd-operator.

Also, see Section 8.5 and Appendix B for many examples of how FDC constraints can be coded in AKL(FD).

8.5 Programming in AKL(FD)

The expressiveness of AKL(FD) is illustrated by examples, ranging from the trivial to those using the full potential of AKL(FD). We exemplify both entailment checking, as well as arc-consistency checking constraints.

Arithmetical constraints such as $x = y + 1$ are executed by conjunctions of indexicals which are either defined by the user/system or generated by the compiler.

The traditional method of constrain-and-generate that is prevalent in constraint logic programming is easily adapted to AKL(FD), which is illustrated by the queens/2 program.

The guard operators $\rightarrow$ and $|$ of AKL enable us to implement constraint-propagating symbolic constraints, as is shown by the count/3 and and/3 examples below.

Conditional reasoning can be defined either in terms of the $\rightarrow$ and $|$ operators, or by indexicals with conditional ranges, as exemplified by the eqjff/3 predicate below.

Furthermore, we exploit the reasoning of Table 3.1 for entailment checking and for controlling the execution of indexicals. For example, in the eqjff/3 program we use antimonotone indexicals to check the entailment of arithmetical equalities, as well as nonmonotone indexicals to suspend the computation of some constraints.

Programming with nonmonotone expressions is unorthodox in the world of constraint programming. However, we claim that since it is possible to implement monotonicity checking of indexicals efficiently (Chapter 10), and since the implementation of Table 3.1 can be made efficient (Section 6.3), there is a large potential in programming with nonmonotone indexicals.

Indexicals can be used for implementing constructive reasoning with disjunctions, exemplified by the element/3 program below. This is possible using the range operators $\lor$ and $\Rightarrow$, basically translating disjunctions to unions of conditional ranges.

AKL(FD) makes different formulations of the same constraint possible. We see this as an asset of the language, since different formulations have different efficiency and different deductive power. The programmer can thus experiment with alternatives, tailoring the constraints to fit the particular application.

User-defined constraints have been recognized as crucial for the versatility of a constraint programming language [DC93b, HSD91, ECR93, ILO93].
We claim that AKL(FD) is a language where complex finite domain constraints can be defined purely within the concurrent constraint framework, and still execute efficiently both when used for consistency checking and when used for entailment checking.

8.5.1 Example: arithmetics

First we show how a simple constraint such as $x = y + 1$ can be defined in AKL(FD).

```prolog
'x=y+1'(X,Y) :-
  fd( X in dom(Y)+1,
       Y in dom(X)-1 ).
```

Hence, $X$ is constrained by the set of possible values of $Y$ pointwise incremented by 1, and $Y$ is constrained by the set of possible values of $X$ pointwise decremented by 1. Since $X = Y + 1$ is linear, the constraint can be compiled by the AKL(FD) compiler for both consistency and entailment checking.

Compiling $X = Y + 1$ for consistency checking results in the indexicals

```
X in min(Y)+1 .. max(Y)+1,
Y in min(X)-1 .. max(X)-1.
```

The compiler generates interval reasoning indexicals. Compiling $X = Y + 1$ for entailment checking generates the antimonotone indexical

```
1 in max(X)-min(Y)..min(X)-max(Y)
```

which succeeds when $X - Y = 1$ is true.

8.5.2 Example: n-queens

The CLP technique of first constraining the solution set and then searching through it can be used in AKL(FD). The well-known $n$-queens program is simply coded as

```
queens(N,L) :-
  domain(L, 1, N),
  constrain(L),
  enum(M, L). % M is preferably either ff or copy_ff

constrain([]) :-
  true.
constrain([D|R]) :-
  constrain_each(R, D, 1),
  constrain(R).
constrain_each([], D, S) :-
```


\[ \rightarrow \text{true.} \]

\[
\text{constrain}_{\exists\!}\text{ach}(E|R), D, S) :-
\rightarrow \text{no}\_\text{threat}(D, E, S),
\text{constrain}_{\exists\!}\text{ach}(R, D, S+1).
\]

\[
\text{no}\_\text{threat}(X, Y, N) :-
\text{fd}(X \in \neg \text{dom}(Y) \land -\text{dom}(Y)+N \land -\text{dom}(Y)-N),
Y \in \neg \text{dom}(X) \land -\text{dom}(X)+N \land -\text{dom}(X)-N).
\]

Since the expression \(-\text{dom}(Y)(\pm N)\) is not monotone in all stores, the effect of the formulation is that \text{no}\_\text{threat}/3 suspends until either X or Y is determined, thus making the two constraints monotone. Hence, we achieve the intended propagation of \(\neq\) [Hen89] through the monotonicity reasoning depicted in Table 3.1. Note that the same constraint in clp(FD) is defined by the use of a special \text{val} range function [DC93b], which, thus, we do not need.

Also note that \text{enum}/2 can be called anywhere in the \text{queens}/2 clause, since the stability condition of AKL guarantees that all determinate work is performed before any nondeterministic step is taken. Hence, the search is not initiated until all constraint propagation is completed.

\text{no}\_\text{threat}/3 can alternatively be defined by

\[
\text{no}\_\text{threat}(X, Y, N) :-
\text{fd}(X \neq Y),
\text{fd}(X \neq Y+N),
\text{fd}(X \neq Y-N)
\]

which is a less efficient formulation, since three constraints are used instead of one.

### 8.5.3 Example: atmost

In constraint logic programming systems such as CHIP, cardinality reasoning is performed by builtins. We give an example of how cardinality reasoning can be programmed in AKL(FD), using the committed choice operator. The cardinality combinator can similarly be defined in AKL(FD) by using the abstraction notion of AKL [JMB94], thereby calling the constraints as predicates.

Consider the constraint \text{atmost}(u, l, v) which is true iff at most \(u\) elements in \(l\) are equal to \(v\), where \(l = [x_1, \ldots, x_k]\). The constraint can be defined by the formula

\[
\sum_{i=1}^{k} (b_i) \leq u
\]

where \((b_i)\) is 1 iff \(x_i = v\) is true and 0 iff \(x_i \neq v\) is true. The following is an encoding of the formula in AKL(FD).
atmost(U, L, V) :-
    fd(N in 0..U),
    count_V(N, L, V).

count_V(N, [], .) :-
    -> fd(N in 0).

count_V(N, [X|L], V) :-
    -> fd(B in 0..1),
        eqiff(X, V, B),
        fd(B+M=N),
        count_V(M, L, V).

eqiff(X, V, B) :- fdask(X \neq V) | fd(B in 0).

eqiff(X, V, B) :- fdask(X=V) | fd(B in 1).

eqiff(X, V, B) :- fdask(B=0) | fd(X \neq V).

eqiff(X, V, B) :- fdask(B=1) | fd(X=V).

Note the use of the commit operator, which is crucial to the functionality of the predicate. The test X \neq V may well suspend in a situation where B=0 succeeds, or vice versa. The use of -> could unnecessarily suspend the constraint.

eqiff/3 can alternatively be defined using ⇒ as, where we assume V is a constant:

eqiff(X, V, B) :-
    fd(X in ((dom(B) ∧ 1) ⇒ V) ∨
        ((dom(B) ∧ 0) ⇒ -V),
        B in ((dom(X)∧V ⇒ 1) ∨
        ((dom(X)\{V} ⇒ 0)).

Note that the indexicals of the latter eqiff/3 behave exactly as the range functions X ⇒ B and B ⇒ X of clp(FD) [DC93b] by the combination of conditional ranges and monotonicity reasoning. Also, these indexicals are an optimized version of what the indexical scheme in Chapter 4 would generate when applied to X = V ⇔ B = 1 and X \neq V ⇔ B = 0.

The entailment of atmost/3 is defined by atmost_ask/3

atmost_ask(U, L, V) :-
    count_V(N, L, V),
    atm_ask(N, U).

atm_ask(N, U) :-
    fdask(N \leq U)
    | true.

atm_ask(N, U) :-
    fdask(N > U)
    | false.

and replacing eqiff/3 by
eq_iff(X, V, B) :- fd_ ask(X</span>\not=V) | fd( B in 0 ).

or alternatively by

eq_iff(X, V, B) :-
    fd( B in (\langle \text{dom}(X) \rangle \\& \not=V) \Rightarrow 1) \lor
    (\langle \text{dom}(X) \rangle \\& 0 )).

That is, atmost(U, L, V) succeeds when at most U equations $X_i=V$ are true, where $L=[X_1, \ldots, X_k]$.

### 8.5.4 Example: element

We now consider the definition of element/3 [DSH88]. The constraint applies disjunction constructively which can be captured by the use of the FD operators $\Rightarrow$ and $\lor$. The constraint element($i$, [$x_1, \ldots, x_k$], $v$) is true iff $\bigvee_{j=1}^k (i = j \land x_j = v)$ is true, i.e.,

\[
i \in \{ j : x_j \in \text{dom}(v) \} \land v \in \{ x_j : j \in \text{dom}(i) \}
\]

where we assume $x_j$ is a constant, $1 \leq j \leq k$. Note that this definition gives the same pruning as the built-in element/3 of CHIP and clp(FD) does [DSH88, DC93b].

We derive the following program, this time using the $\Rightarrow$ and $\lor$ operators of FD.

\[
\text{element}(I, L, V) :-
    \text{constrain}_I(L, 1, V, I),
    \text{constrain}_V(L, 1, I, V).
\]

\[
\text{constrain}_I([], K, V, I) :-
    I \leftarrow 0 .
\]

\[
\text{constrain}_I([X|L], K, V, I) :-
    K1 \leftarrow K+1,
    \text{constrain}_I(L, K1, V, I0),
    \text{fd}( I \text{ in } (\langle X \rangle \\& \text{dom}(V)) \Rightarrow K ) \lor \text{dom}(I0) .
\]

\[
\text{constrain}_V([], K, J, V0) :-
    \text{fd}( V0 \text{ in } \infty ) \quad \% \infty \text{ is a large constant outside } \text{dom}(V)
\]

\[
\text{constrain}_V([X|L], K, I, V0) :-
    K1 \leftarrow K+1,
    \text{constrain}_V(L, K1, I, V0),
    \text{fd}( V \text{ in } (\langle \text{dom}(I) \rangle \\& K) \Rightarrow X ) \lor \text{dom}(V0) .
\]

A more general formulation can be used for $i$ and $v$ above, i.e. as

\[
i \in \{ j : \text{dom}(x_j) \cap \text{dom}(v) \neq \emptyset \} \land v \in \cup \{ \text{dom}(x_j) : j \in \text{dom}(i) \},
\]
which is given by replacing the expression \((X \land \text{dom}(V))\) in constrains with the expression \((\text{dom}(X) \land \text{dom}(V))\), and by replacing \((\text{dom}(I) \land K) \Rightarrow X\) in constrains with \((\text{dom}(I) \land K) \Rightarrow \text{dom}(X)\). This is a proper extension to the traditional element, such that the elements of the list need not be determined at evaluation time.

### 8.5.5 Example: and

A finite relation can naturally be defined as a set of tuples. However, in many cases the relation maintains a certain relationship between different arguments, e.g. consider boolean and for which \(\text{and}(x, y, 1)\) is true only if \(x = y = 1\). Such internal relationships introduce conditional reasoning which can be used for constraint propagation. We now show an example of how the commit operator of AKL can be used for the conditional reasoning of finite relations.

We define \(\text{and}(x, y, z)\), which is true iff \(x \land y = z\), as

\[
\begin{align*}
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(X=0) \mid \text{fd}(Z=0). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(Y=0) \mid \text{fd}(Z=0). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(X\neq Z) \mid \text{fd}(Y=0). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(Y\neq Z) \mid \text{fd}(X=0). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(Z=1) \mid \text{fd}(X=1, Y=1). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(X\neq Y) \mid \text{fd}(Z=0). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(X=1) \mid \text{fd}(Y=Z). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(Y=1) \mid \text{fd}(X=Z). \\
\text{and}(X, Y, Z) & \leftarrow \text{fdAsk}(X=Y) \mid \text{fd}(X=Z).
\end{align*}
\]

This definition is to be compared with the similar definitions of \(\text{and}\) in clp(FD) definition, and in cc(FD).

In clp(FD), \(\text{and}(x, y, z)\) is defined by a conjunction of indexicals, and only domain constraints of the kind \(x, (x \in \{0\} \Rightarrow x \in \{1\})\) can be propagated [DC93a], which are weaker than constraints of the type \(x = y\). Of course, the version of clp(FD) specialized to boolean indexicals achieves very efficient propagation of domain constraints [DC93b], however, the point we want to make is that in AKL(FD) more general constraint propagation can be implemented.

In cc(FD), \(\text{and}(x, y, z)\) is defined by a conjunction of implications, \((X=0 \Rightarrow Z=0, \ldots, X=Y \Rightarrow X=Z)\), similar to the clauses above (omitting the \(\text{fd}\) and \(\text{fdAsk}\) operators) [HSD92b]. Thus, the same constraints are propagated as in AKL(FD). However, in cc(FD) there is a problem of propagating redundant copies of constraints.

Consider, for example, the atom \(\text{and}(X, Y, 1)\). In AKL(FD) this atom is replaced by \(\text{fd}(X=1, Y=1)\). In cc(FD) it is replaced by \((X=0 \Rightarrow Z=0, \ldots, X=Y \Rightarrow X=Z)\), which is further reduced to \((X=1, Y=1, X=1 \Rightarrow Y=1, Y=1 \Rightarrow X=1, X=Y \Rightarrow X=1)\), which is finally replaced by \((X=1, Y=1, Y=1,\)
8.6 Disjunction in AKL(FD)

We now describe the different kinds of disjunctive programming that can be exploited in AKL(FD). In Section 10.2.2 we evaluate the different schemes proposed. We use two simple constraints as illustrating examples.

Take the member\((n,[x_1,\ldots,x_k])\) constraint, where \(n\) is assumed to be a constant, and \(x_i\) a domain variable. Such constraints are useful surrogate constraints in graph coloring problems.

First, consider its interpretation:

\[
  n = x_1 \lor \cdots \lor n = x_k
\]

This indicates several strategies for its evaluation. It can be evaluated speculatively, hence, guessing successively on \(n = x_1, n = x_2, \) and so on. However, this is in general not a good idea.

Better, it can be evaluated by reasoning with cardinality, i.e. the constraint continuously removes false disjuncts using entailment checking until only one disjunct remains, \(n = x_i\) say, which thereby is added to the store.

Finally, it can be applied constructively, i.e. inconsistent disjuncts are continuously removed, and domain constraints implied by all remaining alternatives are lifted.

The other example of disjunctive constraint is a resource constraint used in the squares example. Namely,

\[
  (b = 1 \land x \in \{p - s + 1,\ldots,p\}) \lor (b = 0 \land x \in \{0,\ldots,p - s\} \cup \{p + 1,\ldots\})
\]

where \(x\) is a domain variable, and \(p\) and \(s\) are constants. The constraint associates the boolean variable \(b\) with whether the domain of some task \(x\) with duration \(s\) covers \(p\), which is some point in space (or time). Thereby, \(b\) can be used to accumulate the aggregated resources which must obey the capacities of the problem.

8.6.1 Speculative disjunction

First we consider speculative disjunction. Assume the following definition is given:

```
member(N, [Y|\_]) :-
  ? fd(Y in N).
member(N, [_|L]) :-
  ? member(N, L).
```
The speculative reading of \( member(n, [x_1,\ldots,x_k]) \) is written as follows:

\[
\begin{align*}
\text{spec-member}(N, L) \leftarrow \\
\quad \text{member}(N, L) \\
\quad \text{? true.}
\end{align*}
\]

Here, the use of deep guards is crucial, where the call to \( \text{member}(N, L) \) spawns a local hierarchy of stores, each corresponding to a disjunct \( n = x_i \), for some \( i \). The runtime behavior is that the leftmost remaining alternative, \( n = x_i \), say, is nondeterministically chosen at a stable point in the computation, where the other alternatives will be considered upon backtracking.

The speculative reading of the resource constraint

\[
(b = 1 \land x \in \{p-s+1,\ldots,p\}) \lor (b = 0 \land x \in \{0,\ldots,p-s\} \cup \{p+1,\ldots\})
\]

is defined as:

\[
\begin{align*}
\text{res-spec}(X, S, P, B) \leftarrow \\
\quad \text{fd}(B\text{ in }1), \\
\quad \text{PS is } P-S+1, \\
\quad \text{fd}(X\text{ in }PS.P) \\
\quad \text{? true.}
\end{align*}
\]

\[
\begin{align*}
\text{res-spec}(X, S, P, B) \leftarrow \\
\quad \text{fd}(B\text{ in }0), \\
\quad \text{PS is } P-S, \\
\quad \text{P1 is } P+1, \\
\quad \text{fd}(X\text{ in }0..PS\lor(P1..)) \\
\quad \text{? true.}
\end{align*}
\]

Its runtime behavior is similar to the behavior of \( \text{spec-member}/2 \), and its performance is inferior compared to the constructive versions (Section 10.2.2).

### 8.6.2 Cardinality-based disjunction

The cardinality version of \( member/2 \) is written using the \( \text{atleast}/3 \) primitive (Section 8.5.3), where \( \text{atleast}(n, [x_1,\ldots,x_k], v) \) is true iff at least \( n \) elements among \( [x_1,\ldots,x_k] \) equal \( v \). Hence, \( member(n, [x_1,\ldots,x_k]) \) is defined by

\[
\text{card-member}(N, L) \leftarrow \text{atleast}(1, L, N).
\]

At runtime \( \text{card-member}(N, L) \) suspends until at most one consistent equation, \( n = x_i \), say, remains, which thereby is added as \( \text{fd}(n = x_i) \) to the store.

The cardinality version of

\[
(b = 1 \land x \in \{p-s+1,\ldots,p\}) \lor (b = 0 \land x \in \{0,\ldots,p-s\} \cup \{p+1,\ldots\})
\]
is thus defined by
\[ \#(1, [b = 1 \land x \in \{p-s+1, \ldots, p\}, b = 0 \land x \in \{0, \ldots, p-s\} \cup \{p+1, \ldots\}], 2) \]
which behaves as first suspending until either \( \neg(b = 1 \land x \in \{p-s+1, \ldots, p\}) \)
or \( \neg(b = 0 \land x \in \{0, \ldots, p-s\} \cup \{p+1, \ldots\}) \) is entailed, thereby executing
\[ \text{fd}(b \text{ in } 0, x \text{ in } 0, p-s \lor p+1..) \] (or \( \text{fd}(b \text{ in } 1, x \text{ in } p-s+1..p) \)). This version propagates less than disjunction applied constructively does, even using the weak constructive reading defined in the next section.

However, the constraint performs better by first rewriting it into
\[ (b = 1 \iff x \in \{p-s+1, \ldots, p\}) \land (b = 0 \iff x \in \{0, \ldots, p-s\} \cup \{p+1, \ldots\}) \]
and consequently defining it as:

```prolog
res_card(X, S, P, B) :-
    fd(ask, B=0 )
    | PS is P-S,
    | P1 is P+1,
    | fd( X in (0..PS)\(\lor\)(P1..) ).
res_card(X, S, P, B) :-
    fd(ask, B=1 )
    | PS is P-S+1,
    | fd( X in PS..P ).
res_card(X, S, P, B) :-
    PS is S-1-P,
    fd(ask, 0 in max(X)-P..min(X)+PS)
    | fd( B = 1 ).
res_card(X, S, P, B) :-
    PS is S-P,
    P1 is P+1,
    fd(ask, 0 in (0..min(X)-P1) \(\lor\) (max(X)+PS ..) )
    | fd( B = 0 ).
```

which is more efficient in this case, since only one arithmetic constraint needs to be entailed before \( \text{res_card} \) commits. Thus, \( \text{res_card}(X, S, P, B) \) suspends either until \( B \) becomes 1 (0), and, thereby, \( X \) is constrained correspondingly, or until \( X \) is constrained to cover \( P \) (not to cover \( P \)), when \( B \) is constrained correspondingly.

### 8.6.3 Weak constructive disjunction

The weak constructive reading of the member/2 disjunction is achieved by using the indexical coding of disjunctions (Chapter 4). This is coded in AKL(FD) by associating a boolean variable \( b_i \) with the consistency of each disjunct \( n = x_i \), and constraining \( x_i \) to \( n \) when the sum of the boolean variables \( b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_k \) equals 0.
weak_member(M, L) :-
    wcd_mem(bools(L, M, Bs),
    wcd_mem(L, [], Bs, M).

wcd_mem(bools([], Bs)) :-
    Bs=[].

wcd_mem(bools([X L], M, Bs)) :-
    fd(B in ((dom(X)\M) 1) \ 0 ),
    Bs=[B|B0],
    wcd_mem(bools(L, M, B0).

wcd_mem([], Bs) :-

sum_up(R, S1),
sum_up(Bs, S2),
fd( S = S1+S2 ),
fd( X in ((dom(S)\A1) 0..) \ 0 ),
sum_up(L, [B|R], Bs, M).

sum_up([], S) :-
    S=0.

sum_up([X L], S) :-
    sum_up(L, S0),
    fd( S = X+S0 ).

In this case, the weak reading adds no pruning except for what is given by
the cardinality-based reading. The behavior of weak_member/2 is that of
spec_member/2.

More interestingly, the weak but constructive version of

$$(b = 1 \land x \in \{p - s + 1, \ldots, p\}) \lor (b = 0 \land x \in \{0, \ldots, p - s\} \cup \{p + 1, \ldots\})$$

is defined as (using the indexical scheme in Section 4.2):

res_weak(X, S, P, B) :-
    PS is P-S,
    PS1 is PS+1,
    P1 is P+1,
    fd( T1 in (PS1..P ),
    fd( T2 in (0..PS)\(P1..) ),
    fd( B in ((dom(T1)\(dom(X)) 1) \ 0 \ ((dom(T2)\(dom(X)) 0),
        X in ((dom(B)\0) \(dom(T2)) \ ((dom(B)\A1) \(dom(T1)) ).

The behavior of res_weak(X, S, P, B) is such that as soon as either B is
constrained to be 1 or 0, X is constrained correspondingly, or when X only
covers P, or no longer covers P, B is constrained correspondingly. Hence,
the behavior is similar to res_card/4 above.
8.6.4 Constraint lifting disjunction

By using constraint lifting we can implement the member/2 constraint as:

\[
\text{lift_member}(N, L) :- \\
\quad \text{member}(N, L) \\
\quad \_ \\
\text{true}.
\]

Thus, the call to member(N,L) spawns a hierarchy of stores to which constraint lifting is applied. Any inconsistent alternative, corresponding to \( n = x_i \) say, is removed, and for all remaining alternatives, domain constraints are lifted. Note that deep guards are crucial in this.

The constraint lifting version of

\[
(b = 1 \land x \in \{p - s + 1, \ldots, p\}) \lor (b = 0 \land x \in \{0, \ldots, p - s\} \cup \{p + 1, \ldots\})
\]

is similarly defined as:

\[
\text{res_lift}(X, S, P, B) :- \\
\quad \text{fd}(B \text{ in } 1), \\
\quad \text{PS is } P-S+1, \\
\quad \text{fd}(X \text{ in } PS..P) \\
\quad \_ \\
\text{true}.
\]

\[
\text{res_lift}(X, S, P, B) :- \\
\quad \text{fd}(B \text{ in } 0), \\
\quad \text{PS is } P-S, \\
\quad P1 \text{ is } P+1, \\
\quad \text{fd}(X \text{ in } 0..PS\lor(P1..) ) \\
\quad \_ \\
\text{true}.
\]

The constraint lifting version of the disjunctions prunes more than any other version, however, at a cost. Constraint lifting is space consuming, and hard to implement efficiently for finite domain constraints (Section 10.2.2).

8.7 Conclusion

We have presented AKL(FD), a deep-guard concurrent constraint programming language for finite domain constraints. By using entailment-checking in FD and the choice statements of AKL, including constraint lifting, or the new condition combinator for FD, powerful symbolic constraints can be defined in the spirit of the glass-box approach of cc(FD). We extend the programming models of cc(FD) by allowing guarded clauses, and different kinds of disjunctive programming.
The implementation of AKL(FD), called AGENTS hereafter, is described in detail. This includes an efficient adaption of the FD solver for hierarchical constraint stores, the instantiation of the generic constraint interface of AGENTS with FD, the search primitives and the compiler extensions added to AGENTS, entailment checking, constraint lifting, the constraint compiler, which, for example, compiles arithmetic constraints to library calls in AGENTS, and the AGENTS libraries for arithmetic and symbolic finite domain constraints. This chapter is an extended version of [CJ95].

9.1 The FD Interface

The interface between FD and AGENTS consists of three parts; the constraint variable, i.e. the domain variable, the indexical, and the constraint, used for managing the hierarchy of stores. In the following we use C-syntax to describe the structures involved.

The major point of variables and constraints in AGENTS is that they must be tagged with an identifier of the local store they belong to, i.e. in which store they were created. Thus, all reasoning with variables and constraints must consider their locality, e.g. when pruning a domain variable, if the variable is external in the store in which it is pruned, the pruning must be added to the constraint list of the local store. Thereby, deinstalling the local store will remove the pruning.

9.1.1 Indexicals

Each indexical is tagged with its locality, i.e. the store in which it is created. An indexical $x \text{ in } r$ is thus represented as (Section 6.4.2):
9.1 The FD Interface

struct indexical {
    Term x;
    long c;
    unsigned long info;
    id *env;
    Term ent;
    argvec args;
    monstruct *moninfo;
}

where x is a reference to x, c is an index to the byte code representing r, info contains information about whether the indexical is used for entailment checking, whether the indexical is part of an inconsistent local store, and when the indexical was last executed. The field env is the identity of the store in which the indexical was added, and ent is a variable, used as an entailment flag. When the indexical is entailed, the flag is unified with 0. Hence, by allocating the variable in the same store as x, the unification constraint of AGENTS [Jan94] guarantees that the binding of the variable is properly deinstalled and installed such that a locally entailed indexical does not incorrectly appear entailed in an external store.

Furthermore, args is a vector of the arguments to the indexicals, and moninfo contains decision information used for deciding whether the indexical is monotone or antimonotone. Ranges are compiled as in Section 6.7.

9.1.2 Variables

A finite domain variable is an instance of the constraint variable of AGENTS, thus represented as:

struct finDom {
    struct varMethod *method;
    id *env;
    suspension *asusp;
    suspension *msusp;
    fd_suspension *isusp;
    struct finDom *next;
    id *trailed;
    unsigned long info;
    unsigned long min;
    unsigned long max;
    bitmask *d;
}

The asusp field contains a list of all builtin AKL calls waiting for the variable to be determined. For example, a call such as X is Y+1, where Y is a finite domain variable suspends until Y is determined.
The `msusp` field contains a list of member constraints (see below) that are suspended on the variable. When pruning an external variable, a member constraint is added to the local store, representing the pruning.

The `isusp` field contains a list of suspension records of the kind `{int prop; indexical *f;}`, where `prop` encodes the propagation dependency (Section 6.2).

Furthermore, `next` is used for queueing the variable into the propagation queue when pruned, hence no extra propagation queue is needed, and `trailed` is the identity of the store where the variable was last trailed (initially set to NULL). The field `info` contains information about whether the variable is currently an interval or a set, what the last pruning consisted of, i.e. whether the maximum value, the minimum value, both, or neither was pruned, when the last pruning occurred, and how large the representation of the domain is. The field `min` (`max`) contains the current minimum (maximum) of the variable, and `d` contains an array of bitvectors representing the domain (set to NULL if the variable is an interval).

### 9.1.3 Constraints

Integrating FD with AGENTS requires certain constraint objects, necessary to maintain the hierarchy of stores. There are two such constraint objects defined: the indexical constraint and the member constraint.

The *indexical constraint* is defined as:

```c
struct fd_constraint {
    constraintMethod *method;
    struct constraint *next;
    indexical *f;
}
```

simply encapsulating the indexical with the necessary interface functions. *Installing* an indexical constraint is done by applying the solver to the referred indexical in the store being installed. *Deinstalling* an indexical constraint is a void operation, and *promoting* an indexical constraint moves the constraint to the store being promoted into.

The *member constraint* is used for declaring a local pruning of some variable, and is defined as:

```c
struct member_constraint {
    constraintMethod *method;
    struct constraint *next;
    Term x;
    unsigned long info;
    unsigned long min;
    unsigned long max;
    id *trailed;
}```
Adaption of the FD Solver

The FD solver needs to be adapted for hierarchical constraint stores. Basically, this involves checking the locality of each indexical before executing it. However, locality testing amounts to tree searching, and must be optimized to avoid a performance penalty.

9.2.1 Locality testing

Before executing an indexical its locality is computed. Only if the indexical belongs to an ancestral store or to the local store it should be executed. In other cases the indexical may belong to a child store, hence that store must be installed before executing the indexical, or the indexical may belong to a sibling store, and hence it should be ignored, or the indexical belongs to an inconsistent store elsewhere in the hierarchy, and hence it should be dismissed.

By marking the path from the root of the hierarchy to the current store, checking whether a given store is an ancestor of the current local store, which is a common case, can be done efficiently. Whenever the locality of a store identity is computed, first it is checked if local, then it is checked if marked, and otherwise the tree is searched to decide whether it identifies a sibling store or a child store.

9.2.2 Installation and suspension

The mechanism for suspending an indexical \( f \equiv x \in r(y) \) is adapted to the store hierarchy. Suppose \( f \) is suspended in store \( \sigma' \), and that \( y \) is external in \( \sigma' \), and local to \( \sigma \) say. Then \( f \) is suspended in \( \sigma' \) as in Section 6.1, and also added to the constraint list connected to \( \sigma' \). Suppose now \( y \) is pruned in \( \sigma \). Then \( f \) is not executed in \( \sigma \) but instead \( \sigma' \) is installed. Thus, \( f \) is executed through the installation, thereby continuing the execution in \( \sigma' \) as appropriate.
9.2.3 Indexical propagation

The propagation of indexicals in AGENTS is defined as an extension of Algorithm 6.3 (Section 6.3). The extension is an adjustment to the store hierarchy. We assume that to each store $\sigma$ in the hierarchy there is a corresponding set $F_\sigma$ of indexicals associated. The condition on fixed point computations is first of all updated.

**Fixed-point Condition:** Suppose we are given a tree (hierarchy) of constraint stores and a selected node $\sigma_l$. Let $\sigma_c$ contain $\sigma_l$ and all ancestor stores of $\sigma_l$. Thereby, the FD solver is a fixed point algorithm which terminates when for each monotone $f$ in $F_{\sigma_c}$, $\sigma_c$ has been updated to a fixed point of $f$, or to be inconsistent with $f$.

Furthermore, since local constraints are only part of the local state, and should not be visible to external computations, care must be taken when pruning external variables.

**Pruning Condition:** Any domain constraint $x \in I$ added to $\sigma_c$ by the fixed point computation, where $x$ belongs to an ancestor of $\sigma_l$, must be protected by a member constraint associated with $\sigma_l$, since it represents a local, conditional pruning, and should not be visible outside $\sigma_l$.

Let $Q$ be a finite queue/set of variables, all stamped with the time of their queueing, $T$ be a (global) time stamp, and let $\sigma$ be a constraint store, initialized by $\sigma_c$.

**Algorithm 9.2.3:**

```plaintext
aklfd_check(Q, \sigma)
{
    while Q not empty do {
        set Q = Q \ {y};
        let $F$ be the set of indexicals suspended on $y$;
        for each $f \equiv x$ in $r$ in $F$ do {
            if $f$ marked dead then dismiss $f$ and continue;
            if $f$ marked entailed then dismiss $f$ and continue;
            if $r$ not affected by the pruning of $y$ then continue;
            if (stamp of $f$ $\geq$ stamp of $y$) then continue;
            let $l$ be the locality of $f$;
            case $l$ of {
                DEAD:
                    mark $f$ as dead and continue;
                CHILD:
                    install store of $l$;
                    continue;
                SIBLING:
                    continue;
                LOCAL:
                ANCESTOR:
                    let $I = x_\sigma \cap r_\sigma$;
            }
        }
    }
}
```
9.2 Adaption of the FD Solver

```plaintext
set stamp of \( f \) equal to \( T \);
case \( I \) of {
\emptyset :
    if \( r \) monotone in \( \sigma \) then fail else continue;
\( x_\sigma \):
    if \( r \) antimonotone in \( \sigma \) then
        mark \( f \) as entailed and dismiss it;
        continue;
    otherwise:
        if \( r \) not monotone in \( \sigma \) then continue;
        increment \( T \);
        set stamp of \( x \) equal to \( T \);
        if \( x \) external to \( \sigma \) declare local pruning \( x \in x_\sigma \);
        set \( \sigma = \sigma \cup \{ x \in I \} \);
        if \( x \) queued then update type of pruning of \( x \);
        set \( Q = Q \cup \{ x \} \);
        wake all member suspensions to stores below \( l \);
        if \( x \) determined then
            wake all agent suspensions to stores below or equal to \( l \);
            if \( r \) constant in \( \sigma \) then
                mark \( f \) as entailed and dismiss it;
            }
}
}
}

return \( \sigma \);

The complexity of the algorithm depends on the number \( n \) of variables, the maximum domain size \( m \) of any variable, the maximum number \( e \) of suspended indexicals of any variable, the cost \( c \) of evaluating a range, and the number \( s \) of stores in the hierarchy.

**Complexity 9.1** The algorithm is \( O(mne) \) in time.

**Proof:** The size of the variable queue is bounded by \( n \). For each dequeueing, at most \( e \) indexicals are evaluated. Before an indexical is evaluated, its locality is determined, which can be done in \( O(s) \) time (the tree of stores may not be balanced). A variable can only be pruned \( m - 1 \) times, i.e. it can be dequeued at most \( m \) times, thus the total number of indexical evaluations is less or equal to \( mne \). Each indexical evaluation is \( O(c) \) in time, i.e. the algorithm is \( O(mne) \) in time.

Note that in the AGENTS-implementation, installing \( l \) is delayed until the propagation in \( \sigma \) has terminated successfully.
9.3 Arithmetic Constraints in AKL(FD)

This section is based on [CC93], and explains the compilation of arithmetic constraints to library calls. We treat the compilation of linear and nonlinear constraints separately.

9.3.1 Linear constraints compiled to library calls

The key idea of this method is to avoid the quadratic code expansion by not generating any indexicals at all, and instead translate each constraint into a sequence of calls to library constraints that maintain arc-consistency. The drawback is that new variables have to be introduced to carry the intermediate values. Since these new variables will take part in the constraint propagation, they will incur a runtime cost.

Compilation steps

The compilation consists of two steps: normalization and decomposition. A similar compilation technique has been exploited in the CHIP system [AB91], but decomposition is not considered. In clp(FD) a similar kind of decomposition is used [Diaz, personal communication].

Normalization. The normalization step simply consists in transforming the arithmetic constraint into its normal form (Section 3.6.1).

Decomposition. The decomposition step performs the actual translation of the relation. The simplest method is simply to compile the left and right hand sides separately and to link them with a binary relation, i.e.

\[ x = S + i \land y = T + j \land x \cdot y \]

A somewhat better approach is to move the constants \( i \) and \( j \) out of the sums and into the linking relation, i.e.

\[ x = S \land y = T \land x \cdot y + (j - i) \]

The latter translation scheme has the advantage that an extra intermediate variable can often be avoided.

The simplest decomposition needs only the six library constraints displayed (see Figure 9.1) (Appendix A), where the first two are used for decomposing the linear expressions and the last four for expressing the arithmetical relation. Special case versions (e.g., for \( i = 0 \)) can be trivially added to this set.

Different strategies for translating \( S \) and \( T \), given a particular set of library constraints, exist. Given the set (see Figure 9.1), \( x = S \) where \( S = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \) could be translated as:
\begin{align*}
a \cdot x &= y \\
x + y &= z \\
x + c &= y \\
x + c &\neq y \\
x + c &\geq y \\
x + c &\leq y
\end{align*}

Figure 9.1: A minimal set of library constraints

\begin{align*}
x_1 + x_2 &= t_1 \\
t_1 + x_3 &= t_2 \\
t_2 + x_4 &= t_3 \\
t_3 + x_5 &= t_4 \\
t_4 + x_6 &= x
\end{align*}

where a change in the \texttt{min} of \(x_1\) will need to propagate through all five generated constraints to reach \(x\). We call this strategy the \textit{linear} decomposition method, since the number of reexecutions (and hence the number of evaluations of \texttt{min} and \texttt{max} attributes) whenever a \texttt{min} or \texttt{max} attribute changes is linear in \(|S|\). An alternative translation of the same expression is:

\begin{align*}
x_1 + x_2 &= t_1 \\
x_3 + x_4 &= t_2 \\
x_5 + x_6 &= t_3 \\
t_1 + t_2 &= t_4 \\
t_3 + t_4 &= x
\end{align*}

where a change in the \texttt{min} of any \(x_i\) will need to propagate through at most three generated constraints to reach \(x\). We call this strategy the \textit{logarithmic} decomposition method, since the number of reexecutions (and hence the number of evaluations of \texttt{min} and \texttt{max} attributes) whenever a \texttt{min} or \texttt{max} attribute changes is logarithmic in \(|S|\).

To further reduce the overhead caused by reexecution and intermediate variables, more library constraints can be introduced so that fewer intermediate variables are needed (e.g., with a library constraint \(a \cdot x + b \cdot y + z = u\) the expression \(S\) above could be compiled with only two intermediates).

There are more compilation options that could be worth considering, but which have not been investigated in detail:
• empirical results show that ordering the elements of $S$ on the coefficients in descending order has some positive effects.

• ordering the generated constraints so as to minimize the creation of free variables (with domain 0..) at runtime. It is usually safe to assume that the source code variables are constrained (by domain declarations) at runtime, so this ordering is trivially obtained.

• ordering the generated constraints so that the runtime behavior (constraint propagation) becomes as efficient as possible. This is quite hard to achieve, and would seem to require global analysis.

Special cases

The compiler can easily recognize several special cases of decomposing the constraint $S + i \cdot T + j$ where $S = a_1 \cdot x_1 + \ldots + a_k \cdot x_k$ and $T = a_{k+1} \cdot x_{k+1} + \ldots + a_n \cdot x_n$:

• if there are no variables, the constraint can be decided immediately,

• unary constraints ($n = 1$) should compile to specialized library constraints,

• a constraint $S = T$ does not need to be linked by $x = y$ at top level, where $x$ and $y$ are the variables that hold the sums of the decomposed expressions $S$ and $T$ respectively; instead, $x$ and $y$ can be unified at compile time.

9.3.2 Nonlinear constraints compiled to library calls

The key idea of this method is, as before, to avoid the quadratic code expansion by not generating any indexicals at all, and instead translate each constraint into a sequence of calls to library constraints that maintain arc-consistency.

The compilation consists of the previously defined two steps: normalization and decomposition (Section 9.3.1), which are trivially extended to nonlinear constraints, thus producing constraints of the form

$$x = S \land y = T \land x \cdot y + (j - i),$$

where $S$ and $T$ are nonlinear sums.

The simplest decomposition needs only the six library constraints previously displayed (see Figure 9.1), plus one more for the nonlinear case (see Figure 9.2) (Appendix A).

The same considerations as for linear constraints with respect to how decomposition is performed apply, as do the special cases.
\[ a \times x \times y = z \]

Figure 9.2: A minimal set of nonlinear library constraints

### 9.4 The AGENTS Compiler

The compiler in AGENTS compiles indexicals, as well as arithmetic constraints used for arc-consistency, entailment checking, and constraint lifting.

Given \( \text{fd}(x_1 \text{ in } r_1, \ldots, x_k \text{ in } r_k) \), the ranges \( r_i \) are compiled as in Section 6.7, \( 1 \leq i \leq k \), saving the byte code with the rest of the program code. Also, for each range \( r_i \), the argument frame \( e_i \), the corresponding dependencies \( p_i, M_r, \) and \( A_r \) are constructed by the compiler.

For each indexical \( \text{fd}(x_i \text{ in } r_i) \) a call to \( \text{mon}_\text{in}(x_i, l_i, v, e_i, p_i, M_r, A_r) \) is generated, where \( l_i \) is unified at runtime with the label of the code for \( r_i \), and \( v \) is a new variable, common to each \( x_i \text{ in } r_i \), \( 1 \leq i \leq k \), used as an entailment flag (Section 6.2). When \( \text{mon}\_\text{in}/7 \) is executed, some runtime type tests are performed, and, if the tests succeed, the indexical is marked as used for arc-consistency, and \( \text{ak1fd}\_\text{check} \) is invoked (Section 9.2). For example, it is checked that \( r_i \) is well-defined before the computation proceeds. If some variables are unconstrained, or undetermined, depending on what is needed, the call to \( \text{mon}\_\text{in}/7 \) suspends until more information is provided.

Given \( \text{fd}(c_1, \ldots, c_k) \), the compiler compiles each \( c_i \) to a sequence of library calls (Section 9.3), or to inline indexicals (Section 4.1.1), used for arc-consistency. The choice is made depending on the arity of the constraints.

Given \( \text{fd}_{\text{ask}}(c_1, \ldots, c_k) \), the compiler compiles the conjunction \( c_1 \land \cdots \land c_k \) for entailment as in Section 5.3, hence, generating an indexical 0 \( \text{in } r \), for some antimonotone range \( r \). Consequently, the indexical is compiled for entailment checking (Section 6.7), and the call \( \text{am}\_\text{in}(0, l_r, v, e_r, p_r, M_r, A_r) \) is generated, where \( l_r, v, e_r, \) and \( p_r \) are defined similarly as above, but where the indexical is marked as used for entailment checking.

The expression \( \text{fd}_{\text{ax}}(c_1, \ldots, c_k) \) is compiled as \( c_1 \parallel \text{true} \ldots \parallel c_n \parallel \text{true} \). In future versions of the compiler, however, also the indexical scheme will be used (Section 4.2).

The abstract machine of AGENTS is an extension of the WAM [War83], with instructions for selecting clauses, saving information on the local stack, constructing terms, guarding the clause, and for executing procedure calls. For a full explanation see [Jan94]. Note that the byte code for indexicals is kept separate from the AGENTS emulator which has no support for FD.
9.5 Search in AGENTS

We describe the general scheme for nondeterminism in AGENTS, as well as an experimental optimization of search using Prolog-style backtracking.

9.5.1 Copying-based search

Nondeterminism is introduced in AKL by the wait guard (Section 7.2). A nondeterministic computation step in AGENTS is performed by copying parts of the store hierarchy. That is, let $A$, $B$, and $C$ be conjunctions of constraints (or suspended calls), $V$ be the local variables of $A$, $W_B$ the local variables of $B$, and $W_C$ the local variables of $C$. Then

$$\exists V. (A \land (\exists W_B. B \lor \exists W_C. C))$$

is split into

$$\exists V. (A \land \exists W_B. B) \lor \exists V'. (A' \land \exists W'_C. C')$$

where $V'$ and $W'_C$ are copies of $V$ and $W_C$, and where $A'$ and $C'$ are copies of $A$ and $C$ (see Figure 9.3).

![Figure 9.3: Choice splitting](image)

The latter formula is further simplified by promotion to

$$\exists V \cup W_B. (A \land B) \lor \exists V' \cup W'_C. (A' \land C').$$

A nondeterministic step is only performed in AGENTS when all determinate computation steps that may affect the step have been applied. This is called the stability principle of AKL [Jan94] (Section 7.1). If several nondeterminate steps are possible, the selection rule selects the smallest stable subtree to split.

The basic nondeterminate primitive in finite domain constraint programming is the labeling predicate, which enumerates a list of domain variables. The labeling procedure is outlined as: Given a list of domain variables $[x_1, \ldots, x_k]$, and an index $i$, initially set to 1, do the following.

1. Let $\text{dom}(x_i) = \{a_1, \ldots, a_n\}$.

2. Apply arc-consistency to any one of $x_i = a_j$, $1 \leq j \leq n$. 

3. For each alternative which is arc-consistent, increment $i$ and go to step 1.

In a Prolog-based finite domain language this comes out as:

```prolog
labeling([]).
labeling([X|L]) :-
    domain_of(X, D),
    member(X, D),
    labeling(L).
```

where `domain_of(x, l)` is true when $l$ is the list of domain elements of $x$, and `member(x,l)` is true when $x$ is a member of $l$ [AAB+93]. Usually, the two are combined into one primitive `indomain/1` [DHS+88, Hen89, DC93b].

A heuristic rule, such as the first-fail principle, for selecting a variable in each iteration may improve the performance dramatically [Hen89].

```prolog
labeling_ff([]).
labeling_ff([X|L]) :-
    select_smallest_domain([X|L], X0, L0),
    domain_of(X0, D),
    member(X0, D),
    labeling_ff(L0).
```

where `select_smallest_domain(l, x, l0)` is true when $x$ is the variable with the smallest domain in $l$, and $l0$ is $l$ removed of $x$.

In AGENTS, the stability of computations must be taken into account to guarantee that all determinate work has been completed before the next step in the labeling is performed. This is solved by making `indomain` a builtin which suspends until a stable state is reached. Furthermore, `indomain` exploits the short-circuit technique of concurrent logic programming [Sar89], to evaluate the rest of the nondeterminate steps lazily.

Thus, the call `labeling([x1, ..., xk])` suspends until the computation state is stable. Then, a value $n$ is selected nondeterministically to assign to $x_1$, and a copying step is executed. Note that this implies that all constraints, indexicals and variables referred to by any one of $x_i, 1 \leq i \leq k$, are copied. The predicate `indomain/3` then applies arc-consistency by calling `fd(x1 in n)` internally.

```prolog
labeling([]) :-
    -> true.
labeling([X|L]) :-
    -> indomain(X, L, L0),
    labeling(L0).
```

```prolog
indomain(X, L, L0) :-
    select_min_when_stable(X, N)
```
-> in_domain(X, N, L, L0).
indomain(X, L, L0) :-
-> L=L0.

in_domain(X, N, L, L0) :-
  ? fd( X in N ),
  L=L0.
in_domain(X, N, L, L0) :-
  ? fd( X in N+1... ),
  indomain(X, L, L0).

where select_min_when_stable(x, n) succeeds, and unifies n with min(x),
when x is not determined and the computation state is stable. If x is
determined, the call fails.

We also provide a first-fail version of labeling, and a version based on
constraint lifting. The first-fail version is standard and is not explained
further. The lifting version is based on applying constraint lifting to the
statement member(x, [n1, ..., nk]), which is equivalent to x = n1 ∨ ... ∨ x =
n + k.

Let domain_of(x,l) compute the list l of elements in the domain of x,
and let || be the constraint lifting guard operator (Section 8.3). Hence,
constructive labeling is defined as:

lift_labeling([], X, l) :-
  member(X, D),
  lift_domain(X, D),
  lift_labeling(L).

An alternative definition of lift_domain/2 can be used which builds a bal-
anced binary tree instead. Let split_list/3 be a predicate which splits a
list in half, which has a straight-forward definition. Then lift_domain/2 is
defined as:

lift_domain(X, []) :-
  || true.
9.5 Search in AGENTS

lift_domain(X, [N]) :-
   fd(X in N)
|| true.
lift_domain(X, [N,M|D]) :-
   split_list(D, L, R),
   lift_domain(X, [N|L]),
   lift_domain(X, [M|R])
|| true.

Note that lift_domain/2 requires a deep-guard language. Finally, note that it is a simple matter to split the domains in subsets instead of into individual elements. This can sometimes be advantageous [Hen89].

9.5.2 Trailing-based search

We have experimented with a Prolog-style backtracking scheme instead of copying parts of the computation state, since copying becomes very expensive in the case of massive graphs of indexicals. Our experiment is done by defining nondeterministic enumeration procedures as builtins which compute their solutions using trailing, and then promote a copy of the result. Hence, they encapsulate the nondeterminate computation, similar to a deep-guard computation.

The enumeration procedures are written in C, where two stacks are used; the choice-point stack and the trail stack. Each choice-point frame contains the variable currently selected for assignment, the next assignment value, the list of the rest of the variables to be enumerated, and a pointer to the top of the trail as it was before the variable was assigned its current value.

The fields next, trailed, info, min, max, d of a finite domain variable are trailed when the variable is pruned. The fields ent and moninfo of an indexical are trailed when the indexical becomes entailed or when its monotonicity information is updated.

Information concerning guards is currently not trailed, but only destructive changes to finite domains and indexicals are. Hence, suspended guards affected by the enumeration are not installed during the backtracking, but only when the enumeration has succeeded.

Member and builtin suspensions from the enumerated variables (Section 9.1) are collected before the enumeration is initiated. Consequently, a solution is computed and a copy is made of the local computation state containing the solution. The solution is then promoted, and the collected guard suspensions are installed.

When selecting the noncopied part of the computation state, the topmost choice-point frame is used for resetting the part of the trail corresponding to the latest choice, and enumeration can be reexecuted. The AKL part of the enumeration is as follows:

labeling(L) :-
Implementation of AKL(FD)

collect_suspensions(L, S),
init_labeling(L, C),
c_labeling(C),
labeling(C, S).

labeling(., S) :-
  ? wake_suspensions(S).
labeling(C, S) :-
  ? c_labeling(C),
  labeling(C, S).

The parameter C denotes the bottom of the choice-point stack, such that several enumerations can be active simultaneously. The builtin procedures init_labeling/2 and c_labeling/1 are defined as follows. 

**Init Labeling**: Given a list \([x_1, \ldots, x_k]\) of finite domain variables. Let \(t\) refer to the top of the trail stack. Initialize the choice-point stack with a watchdog frame \(w\), and push \(c = \langle x_1, \text{\texttt{min}}(x_1), [x_2, \ldots, x_k], t \rangle\).

C-labeling is a procedure which recurs over the list of domain variables, assigning values until failure occurs or until the end of the list appears. At each recursive step a new choice-point is pushed, containing the information necessary to redo the recursive step with another assignment. When failure occurs either another value is chosen for the current assignment, or if no such value exists, the choice-point stack is consulted, using the topmost frame to redo the previous step with another assignment. For each assignment, the variable determined is trailed, and arc-consistency is applied. Any updates caused by the arc-consistency propagation are trailed.

**Algorithm 9.5.2:**

\begin{verbatim}
labeling(c) where c = \langle x, n, l, t \rangle, and
x is a domain variable, n is the next value to assign to x,
l is a list of domain variables, t is a trail reference;
{
  let d = \text{\texttt{dom}}(x);
  for m \in d such that m \geq n {
    if x \text{ in } m fails (using Algorithm 6.3) then
      reset the trail to t, and continue;
    else {
      if l = [] then return SUCCESS;
      if l = \langle y | l' \rangle then
        let t' be the current trail top, replace n in c with m + 1,
        push \(c' = \langle y, \text{\texttt{min}}(y), l', t' \rangle\), and return labeling(c');
    }
  }
  pop c, and let \(c'\) be the previous choice-point frame;
  if \(c' = w\) then return FAIL;
  let \(c' = \langle x', n', l', t' \rangle\);
\end{verbatim}
reset the trail to $t'$, and return labeling($c'$);
}

**C Labeling:** Suppose the top of the choice-point stack equals $c = \langle x, n, l, t \rangle$. Reset the trail to $t$, and return labeling($c$).

Hence, given a list of domain variables, $l$, initializing the stacks with **Init Labeling**, and repeatedly calling **C Labeling** will exhaust the solution space of $l$.

As a final note; combining **C Labeling** with Algorithm 6.3, the emulator in Section 6.6, the compilation techniques in Section 6.7 and Chapter 4, actually forms a kernel for finite domain constraint solving which can be integrated in many programming languages, even non-logical such. For example, we are currently using the above as a basis for adding FD to the functional concurrent programming language Erlang [AVW93].

### 9.6 Lifting domain constraints

Finally, we will consider lifting domain constraints in AGENTS. Thus, by the design of constraint lifting in AGENTS (Section 8.3), all we need to do is to provide a function $\alpha$, such that, given the local constraint stores $\sigma_1, \ldots, \sigma_k$, $\alpha(\sigma_1, \ldots, \sigma_k) \subseteq \sigma_i$ holds, for any $i$ between 1 and $k$.

Given the constraint stores $\sigma_1, \ldots, \sigma_k$,

$$\alpha(\sigma_1, \ldots, \sigma_k) = \{ x \in x_{\sigma_i} \cup \cdots \cup x_{\sigma_1} : x_{\sigma_i} \neq N, 1 \leq i \leq k \}.$$  

Obviously, $\alpha(\sigma_1, \ldots, \sigma_k) \subseteq \sigma_i$ holds.

By keeping the local stores sorted on variable names, generating $x \in x_{\sigma_i} \cup \cdots \cup x_{\sigma_1}$, where $x_{\sigma_i} \neq N$, incrementally for $n$ variables can be done in $O(n \log k)$ time. Furthermore, in AGENTS, local stores are associated with a DIRTY-bit which is set initially and whenever a local store can no longer be guaranteed to be sorted (such as after a garbage collection, or after the addition of new constraints). Hence, before lifting is performed for domain constraints, each local store $\sigma_i$ is checked if dirty, $1 \leq i \leq k$. If dirty, the store is sorted and the bit is reset. This improves the incremental behavior of lifting.

**Example 9.6:** Running $x = y$, $x = z$, ($y = 1 \uparrow$ true; $z = 1 \uparrow$ true) will produce the stores $\sigma_y = \{ x \in \{1\}, y \in \{1\}, z \in \{1\} \}$ and $\sigma_z = \{ x \in \{1\}, y \in \{1\}, z \in \{1\} \}$. Hence, $x \in \{1\}$ is lifted and $\sigma$ is updated to $\{ x \in \{1\}, y \in \{1\}, z \in \{1\} \}$.  

### 9.7 Conclusion

The implementation of AKL(FD) adapts the FD algorithms in Chapter 6 to the generic constraint interface of AGENTS and the store hierarchy of AKL. The performance of AGENTS is somewhat degraded by this, compared to a flat implementation (Chapter 10).
Chapter 10

Evaluation

We compare AGENTS, the implementation of AKL(FD), with two other systems. The implementation of FD and the trailing scheme of AGENTS is compared to clp(FD) [DC93c]. We analyze

- the efficiency of the FD solver,
- the effect of the optimizations exploited, and
- the difference in speed between AGENTS and clp(FD).

Furthermore, AGENTS is compared with Oz, since Oz [Smo94, SS94, MMP89] provides the same basic notions for deep guards as AKL does, and we compare their performance with respect to finite domain constraints and guard programming. In this comparison, the copying-based labeling is used.

The evaluation is based on a set of well-known benchmark problems, which are described in Chapter 2.

eq10 A problem involving 10 linear equations (Section 2.1.1). Naïve labeling is used.
eq20 A problem involving 20 linear equations (Section 2.1.2). Naïve labeling is used.

dsendmory The SEND+MORE=MONEY problem in Section 2.1.3. Naïve labeling is used.

alpha The cipher problem in Section 2.1.4. Naïve and first-fail labeling are both used.
five The combinatorial puzzle in Section 2.1.5. Naïve labeling is used.

queens 64 & 96 The 64(96)-queens problem [Hen89] (Section 2.1.6), first solution. First-fail labeling is used.
The Sudoku problem in Section 2.1.7. Naïve labeling is used.

The magic problem of size 50 in Section 2.1.8. Naïve and first-fail labeling are both used.

cars The cars problem in Section 2.3.1. Naïve labeling is used and an optimized version of element3 (Appendix A).

The reprographic machine problem in Section 2.3.2, applied to input #1 in Table 10.8. Naïve labeling is used.

The squares tiling problem in Section 2.2.1, using first-fail labeling.

We also evaluate the different schemes for disjunctions proposed (Section 3.6.5).

10.1 Evaluating AGENTS with Trailing

The FD solver is defined in Chapter 6, and its implementation in AGENTS (Chapter 9) is evaluated by analyzing the effects of the optimizations and how it compares to clp(FD). In this comparison we are using the trailing scheme developed in Section 9.5.2.

The major difference between AGENTS and clp(FD) is that in clp(FD) indexicals and programs are compiled to low-level C, whereas in AGENTS the generic constraint interface is used to call the indexicals which are emulated separately from the AGENTS emulator.

For each problem we measure certain aspects of the execution, namely the evaluation of indexicals, and the work involved in searching for a solution. More precisely, we measure the following for each program run until the first solution is found:

Check The number of times an indexical is executed.

Prune The number of times a variable is pruned.

Useless The number of times the execution of an indexical does not prune a variable.

Fail The number of times an indexical fails.

Mono The number of times the monotonicity algorithm is invoked.

NonD The number of nondeterminate steps that is taken until termination.

Trig The number of times the optimization(s) triggered.

RunT The run-time in milliseconds until termination.

Hence, we are able to give a precise account of the effect of the optimizations.
10.1.1 Evaluating the optimizations

We now evaluate the three optimizations in Section 6.2 which affect the performance of propagation of domain constraints; namely entailment marking, multiple indexical avoidance, and variable dependencies.

Table 10.1 contains performance data for the benchmarks programs run with all optimizations turned off, and Table 10.5 contains the corresponding figures with all optimizations turned on.

The optimization concerning marking entailed indexicals is evaluated in Table 10.2. The second optimization evaluated (Table 10.3) is the one which uses time stamps to filter out multiple occurrences of indexicals in the propagation queue.

<table>
<thead>
<tr>
<th>Program</th>
<th>Check</th>
<th>Prune</th>
<th>Useless</th>
<th>Fail</th>
<th>Mono</th>
<th>NonD</th>
<th>RunT</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq10</td>
<td>6094</td>
<td>545</td>
<td>5500</td>
<td>49</td>
<td>5060</td>
<td>58</td>
<td>810</td>
</tr>
<tr>
<td>eq20</td>
<td>7987</td>
<td>711</td>
<td>7277</td>
<td>49</td>
<td>7791</td>
<td>56</td>
<td>1060</td>
</tr>
<tr>
<td>sendmory</td>
<td>164</td>
<td>54</td>
<td>109</td>
<td>4</td>
<td>1192</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>alpha</td>
<td>679653</td>
<td>151237</td>
<td>520243</td>
<td>8440</td>
<td>144787</td>
<td>9731</td>
<td>3090</td>
</tr>
<tr>
<td>alpha-ff</td>
<td>5714</td>
<td>1029</td>
<td>4674</td>
<td>11</td>
<td>12373</td>
<td>17</td>
<td>340</td>
</tr>
<tr>
<td>five</td>
<td>37</td>
<td>185</td>
<td>210</td>
<td>5</td>
<td>537</td>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>suudoku</td>
<td>2934</td>
<td>583</td>
<td>2347</td>
<td>4</td>
<td>4090</td>
<td>9</td>
<td>210</td>
</tr>
<tr>
<td>queens-f 96</td>
<td>18930</td>
<td>4403</td>
<td>14501</td>
<td>26</td>
<td>406110</td>
<td>135</td>
<td>4570</td>
</tr>
<tr>
<td>magic 50</td>
<td>702321</td>
<td>89002</td>
<td>612844</td>
<td>308</td>
<td>601680</td>
<td>392</td>
<td>3900</td>
</tr>
<tr>
<td>magic-f 50</td>
<td>60171</td>
<td>8391</td>
<td>51773</td>
<td>7</td>
<td>58880</td>
<td>34</td>
<td>270</td>
</tr>
<tr>
<td>cars</td>
<td>2297</td>
<td>381</td>
<td>1916</td>
<td>0</td>
<td>1651</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>rpg 1</td>
<td>279845</td>
<td>34704</td>
<td>213441</td>
<td>1700</td>
<td>890026</td>
<td>2064</td>
<td>12420</td>
</tr>
</tbody>
</table>

Table 10.1: Performance data for solver without optimizations

<table>
<thead>
<tr>
<th>Program</th>
<th>Check</th>
<th>Prune</th>
<th>Useless</th>
<th>Fail</th>
<th>Mono</th>
<th>NonD</th>
<th>Trig</th>
<th>RunT</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq10</td>
<td>5928</td>
<td>545</td>
<td>5334</td>
<td>49</td>
<td>5060</td>
<td>58</td>
<td>106</td>
<td>740</td>
</tr>
<tr>
<td>eq20</td>
<td>7381</td>
<td>711</td>
<td>6771</td>
<td>49</td>
<td>6335</td>
<td>56</td>
<td>156</td>
<td>1000</td>
</tr>
<tr>
<td>sendmory</td>
<td>123</td>
<td>54</td>
<td>68</td>
<td>1</td>
<td>114</td>
<td>2</td>
<td>78</td>
<td>20</td>
</tr>
<tr>
<td>alpha</td>
<td>660799</td>
<td>151275</td>
<td>601064</td>
<td>8440</td>
<td>131676</td>
<td>9731</td>
<td>336202</td>
<td>36750</td>
</tr>
<tr>
<td>alpha-ff</td>
<td>5665</td>
<td>1029</td>
<td>4125</td>
<td>11</td>
<td>1481</td>
<td>17</td>
<td>1892</td>
<td>300</td>
</tr>
<tr>
<td>five</td>
<td>214</td>
<td>138</td>
<td>80</td>
<td>5</td>
<td>330</td>
<td>8</td>
<td>201</td>
<td>40</td>
</tr>
<tr>
<td>suudoku</td>
<td>2934</td>
<td>583</td>
<td>2347</td>
<td>4</td>
<td>4090</td>
<td>9</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>queens-f 96</td>
<td>50033</td>
<td>4403</td>
<td>1174</td>
<td>26</td>
<td>208797</td>
<td>135</td>
<td>37333</td>
<td>3230</td>
</tr>
<tr>
<td>magic 50</td>
<td>244730</td>
<td>89002</td>
<td>612844</td>
<td>308</td>
<td>601680</td>
<td>392</td>
<td>3900</td>
<td></td>
</tr>
<tr>
<td>magic-f 50</td>
<td>42977</td>
<td>8391</td>
<td>51773</td>
<td>7</td>
<td>58880</td>
<td>34</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>cars</td>
<td>1661</td>
<td>381</td>
<td>1280</td>
<td>0</td>
<td>1015</td>
<td>2</td>
<td>630</td>
<td>60</td>
</tr>
<tr>
<td>rpg 1</td>
<td>170465</td>
<td>34704</td>
<td>213441</td>
<td>1700</td>
<td>899719</td>
<td>2604</td>
<td>319007</td>
<td>7940</td>
</tr>
</tbody>
</table>

Table 10.2: Performance data for solver with entailment marking

Note that, in fact, the time stamp optimization may change the number of prunings. Compare, for example, the number of prunings for the 10-equations problem in Table 10.1 and Table 10.3.

To see how this works, consider a multi-arity indexical \( f \equiv x \ in r(y, z) \),
and the propagation queue

\[ Q : y \rightarrow x_1 \rightarrow \cdots \rightarrow x_k \rightarrow z \rightarrow \emptyset. \]

Thus, if the current time stamp is \( i \), it follows that the time stamps of \( y \) and \( z \) are less or equal to \( i \). Suppose now \( y \) is dequeued, and thereby \( f \) is reexecuted, updating the execution time of \( f \) to be \( j \), where \( j \geq i \). Henceforth, when \( z \) is dequeued and \( f \) is considered, the optimization triggers, thereby ignoring \( f \), even if the queue now looks like

\[ Q : y \rightarrow x_1 \rightarrow \cdots \rightarrow y_k \rightarrow y \rightarrow \emptyset, \]

i.e. \( y \) may have been pruned and queued again, before \( z \) is dequeued. This may affect how soon the propagation terminates.

**Example 10.1.1:** Let \( f \) be \( x \in \text{min}(y) + \text{min}(z) \cdot \text{max}(y) + \text{max}(z) \), the time stamps of \( y \) be 10, \( z \) be 5, and the execution stamp of the indexical be 7. Suppose \( \sigma \) is the store \( \{ x \in \{ 1 \}, y \in \{ 1 \}, z \in \{ 1, 2 \} \} \), and \( Q \) is the queue

\[ z \rightarrow w \rightarrow y \rightarrow \emptyset. \]

Note that \( f \) is inconsistent in \( \sigma \).

Hence, when \( z \) is dequeued, \( f \) is not executed because of the optimization. Instead \( w \) is dequeued, and all indexicals constrained by \( w \) are considered for execution. Suppose at least one of these indexicals are executed, and that none of these indexicals fail. Hence, failure occurs when \( y \) is dequeued. However, if \( f \) is executed when \( z \) is dequeued, failure occurs before \( w \) is considered, thereby avoiding to execute the indexicals suspended on \( w \).

Finally, the optimization regarding variable dependencies is evaluated in Table 10.4. The variable dependency optimization may also change the number of prunings. Compare, for example, the number of prunings for the alpha problem in Table 10.1 and Table 10.4.
Table 10.4: Performance data for solver with dependency checking

<table>
<thead>
<tr>
<th>Program</th>
<th>Check</th>
<th>Prune</th>
<th>Useless</th>
<th>Fail</th>
<th>Mono</th>
<th>NonD</th>
<th>Trig</th>
<th>RunT</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq10</td>
<td>6694</td>
<td>545</td>
<td>5609</td>
<td>49</td>
<td>5026</td>
<td>58</td>
<td>0</td>
<td>770</td>
</tr>
<tr>
<td>eq20</td>
<td>7387</td>
<td>711</td>
<td>7227</td>
<td>49</td>
<td>7914</td>
<td>56</td>
<td>0</td>
<td>1030</td>
</tr>
<tr>
<td>sendmory</td>
<td>154</td>
<td>54</td>
<td>99</td>
<td>1</td>
<td>116</td>
<td>2</td>
<td>161</td>
<td>10</td>
</tr>
<tr>
<td>alpha</td>
<td>541089</td>
<td>15189</td>
<td>38160</td>
<td>8440</td>
<td>531259</td>
<td>9731</td>
<td>920621</td>
<td>29330</td>
</tr>
<tr>
<td>alpha-ff</td>
<td>5479</td>
<td>1028</td>
<td>4440</td>
<td>11</td>
<td>5963</td>
<td>17</td>
<td>8289</td>
<td>310</td>
</tr>
<tr>
<td>five</td>
<td>357</td>
<td>158</td>
<td>194</td>
<td>5</td>
<td>311</td>
<td>8</td>
<td>308</td>
<td>40</td>
</tr>
<tr>
<td>sudoko</td>
<td>2934</td>
<td>583</td>
<td>2341</td>
<td>4</td>
<td>2239</td>
<td>9</td>
<td>4308</td>
<td>150</td>
</tr>
<tr>
<td>queens-ff</td>
<td>18930</td>
<td>4403</td>
<td>14501</td>
<td>26</td>
<td>18795</td>
<td>135</td>
<td>387315</td>
<td>2890</td>
</tr>
<tr>
<td>magic-50</td>
<td>702324</td>
<td>89072</td>
<td>612884</td>
<td>308</td>
<td>691680</td>
<td>392</td>
<td>0</td>
<td>37660</td>
</tr>
<tr>
<td>magic-ff</td>
<td>66001</td>
<td>8391</td>
<td>57653</td>
<td>7</td>
<td>50715</td>
<td>34</td>
<td>170</td>
<td>2620</td>
</tr>
<tr>
<td>cars</td>
<td>2280</td>
<td>381</td>
<td>8899</td>
<td>0</td>
<td>3634</td>
<td>2</td>
<td>30</td>
<td>90</td>
</tr>
<tr>
<td>rpg 1</td>
<td>201156</td>
<td>34704</td>
<td>364752</td>
<td>1780</td>
<td>230110</td>
<td>2604</td>
<td>57929</td>
<td>9350</td>
</tr>
</tbody>
</table>

To see how this works, consider a multi-arity indexical \( x \text{ in } r(y, z) \), and the propagation queue

\[
Q : y \rightarrow x_1 \rightarrow \cdots \rightarrow x_k \rightarrow z \rightarrow []
\]

Suppose, for the sake of the argument, that the pruning type of \( y \) is MIN, and the dependency between \( y \) and \( r \) is MAX, and that the pruning type of \( z \) is MAX, and the dependency between \( z \) and \( r \) is MAX. Thus, when \( y \) is dequeued, the optimization triggers and the indexical is ignored, even though, if executed, \( x \) would be pruned because of the change in \( z \). Then, if failure occurs before \( z \) is dequeued, the pruning of \( x \) caused by \( r(y, z) \) will never take place, i.e. the number of prunings is smaller with the optimization turned on than otherwise.

**Example 10.1.1:** Let \( f \) be the indexical be \( x \text{ in } 0 \cdot \max(y) + \max(z) \). Suppose \( \sigma \) is the store \( \{x \in \{0, \ldots, 4\}, y \in \{1, 2\}, z \in \{1, 2\}, w \in \{1, 2\}\} \). Then, assume \( \sigma \) is updated to \( \sigma' = \{x \in \{0, \ldots, 4\}, y \in \{2\}, z \in \{1\}, w \in \{2\}\} \), i.e. \( y \) is pruned of its minimum value, and \( z \) of its maximum value. Thus, let \( Q \) be the queue

\[
y \rightarrow w \rightarrow z \rightarrow [],
\]

where \( w \) occurs in \( 1 \text{ in } \min(w) \). Hence, when \( y \) is dequeued, \( f \) will not be executed because of the optimization. Instead \( w \) is dequeued, and the indexical \( 1 \text{ in } \min(w) \) is executed, which fails. Hence, \( f \) is never executed. However, if \( f \) is executed when \( y \) is dequeued, \( x \) would be pruned to \( \{0, 1, 2, 3\} \), and only after the pruning, \( 1 \text{ in } \min(w) \) fails.

Finally, in Table 10.6 we evaluate the effect of the optimizations by comparing the measured numbers with and without all the optimizations enabled. For example, for the eq10-program, turning on the optimizations decreases the number of times an indexical is executed by 28%.

In conclusion, the optimizations have the following effect.
10.1 Evaluating AGENTS with Trailing

<table>
<thead>
<tr>
<th>Program</th>
<th>Check</th>
<th>Prune</th>
<th>Useless</th>
<th>Fail</th>
<th>Mono</th>
<th>NonD</th>
<th>Trig</th>
<th>RunT</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq10</td>
<td>4381</td>
<td>586</td>
<td>3790</td>
<td>49</td>
<td>4263</td>
<td>58</td>
<td>2243</td>
<td>530</td>
</tr>
<tr>
<td>eq20</td>
<td>6063</td>
<td>713</td>
<td>5301</td>
<td>49</td>
<td>5867</td>
<td>56</td>
<td>2840</td>
<td>816</td>
</tr>
<tr>
<td>sendmory</td>
<td>105</td>
<td>54</td>
<td>50</td>
<td>1</td>
<td>67</td>
<td>2</td>
<td>210</td>
<td>20</td>
</tr>
<tr>
<td>alpha</td>
<td>458341</td>
<td>154148</td>
<td>299073</td>
<td>8440</td>
<td>149104</td>
<td>9731</td>
<td>1003160</td>
<td>27500</td>
</tr>
<tr>
<td>alpha-ff</td>
<td>3867</td>
<td>1073</td>
<td>2783</td>
<td>11</td>
<td>3744</td>
<td>17</td>
<td>11765</td>
<td>280</td>
</tr>
<tr>
<td>five</td>
<td>226</td>
<td>158</td>
<td>63</td>
<td>5</td>
<td>180</td>
<td>8</td>
<td>133</td>
<td>40</td>
</tr>
<tr>
<td>sudoku</td>
<td>1899</td>
<td>588</td>
<td>1312</td>
<td>4</td>
<td>1204</td>
<td>9</td>
<td>4892</td>
<td>130</td>
</tr>
<tr>
<td>queens-ff 96</td>
<td>5603</td>
<td>4403</td>
<td>1174</td>
<td>26</td>
<td>5468</td>
<td>135</td>
<td>400642</td>
<td>2390</td>
</tr>
<tr>
<td>magic 50</td>
<td>232385</td>
<td>88088</td>
<td>142299</td>
<td>308</td>
<td>221741</td>
<td>392</td>
<td>471127</td>
<td>11600</td>
</tr>
<tr>
<td>magic-ff 50</td>
<td>41075</td>
<td>8439</td>
<td>32629</td>
<td>7</td>
<td>30789</td>
<td>34</td>
<td>203241</td>
<td>1300</td>
</tr>
<tr>
<td>cars</td>
<td>1397</td>
<td>581</td>
<td>1246</td>
<td>0</td>
<td>951</td>
<td>2</td>
<td>749</td>
<td>0</td>
</tr>
<tr>
<td>rpg 1</td>
<td>103772</td>
<td>34104</td>
<td>67668</td>
<td>1700</td>
<td>101086</td>
<td>2604</td>
<td>708553</td>
<td>5890</td>
</tr>
</tbody>
</table>

Table 10.5: Performance data for solver with all optimizations enabled

Table 10.6: 100*(optimizations on/optimizations off)

- No effect on the number of failures or nondeterminate steps, since they are only concerned with decreasing the number of times an indexical is executed.

- A dramatic effect on the number of times an indexical is uselessly evaluated, e.g. consider the queens program where 92% of the redundant executions are eliminated.

- A possibly negative effect on the number of prunings as is shown in Example 10.1.1. However, in the general case, the number of prunings may also decrease because of the optimizations (Example 10.1.1). The bottom-line argument is that indexicals are executed in different order depending on what optimizations are active.

- A dramatic effect on the number of times the monotonicity algorithm is evaluated, e.g. consider the queens program where 99% of the evaluations are eliminated.
A good effect on the run-time, decreasing the run-time with 30% in average, with a peak improvement in the magic-series example (70%).

We now go on to analyze the speed of the solver.

### 10.1.2 Comparing with clp(FD)

In this section we compare AGENTS with clp(FD) which is state-of-the-art among finite domain programming systems with respect to speed [DC93c]. In the following we use the trailing scheme for backtracking (Section 9.5) which closely resembles the trailing scheme of clp(FD).

The machine used throughout in the evaluation is a Sun 4/25 (SPARC-station ELC), running SunOs 4.1.2, and all measurements are done on this machine.

In Table 10.7 we give the timings in milliseconds for the same programs run in AGENTS and in clp(FD). If the calls to element/3 (Appendix A) in the cars-program are replaced with calls to element/3, as defined in Section 8.5.4, the performance of the program is degraded by a factor 2.5 [CJH94].

<table>
<thead>
<tr>
<th>Program</th>
<th>AGENTS</th>
<th>clp(FD)</th>
<th>AGENTS/clp(FD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eql0</td>
<td>580</td>
<td>130</td>
<td>4.5</td>
</tr>
<tr>
<td>eq20</td>
<td>800</td>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>sendmory</td>
<td>30</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>alpha</td>
<td>29510</td>
<td>12400</td>
<td>2.4</td>
</tr>
<tr>
<td>alpha-ff</td>
<td>280</td>
<td>180</td>
<td>1.5</td>
</tr>
<tr>
<td>five</td>
<td>40</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>sundoku</td>
<td>160</td>
<td>60</td>
<td>2.7</td>
</tr>
<tr>
<td>queens-ff 96</td>
<td>2390</td>
<td>450</td>
<td>5.3</td>
</tr>
<tr>
<td>magic-ff 50</td>
<td>1300</td>
<td>1000</td>
<td>1.3</td>
</tr>
<tr>
<td>cars</td>
<td>60</td>
<td>60</td>
<td>1.0</td>
</tr>
<tr>
<td>rpg 1</td>
<td>5890</td>
<td>8630</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Table 10.7: AGENTS versus clp(FD) (ms.)

Furthermore, we give plottings of the two systems applied to a series of inputs to the n-queens, the magic series, and the reprographic scheduling programs.

In Figure 10.1 the runtimes in milliseconds are given for AGENTS and clp(FD) applied to the 60-, 61-, and so on up to the 69-queens problem, using first-fail labeling to compute the first solution. As can be seen, the runtimes are stochastic with respect to the number of queens. However, the relation remains constant between the two systems, where the clp(FD)-program is roughly 5 times faster than the AKL(FD)-program.

In Figure 10.2 the runtimes in milliseconds are given for AGENTS and clp(FD) applied to the magic 50-, 51, and so on up to the 59-series problem,
Figure 10.1: Queens plot of AGENTS versus clp(FD)

using first-fail labeling to compute the unique solution. The execution times
grow linearly with the size of the input for each program, and the AKL(FD)-
program runs about 1.3 times slower than the clp(FD)-program.

<table>
<thead>
<tr>
<th>N</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“sdkddkkkdkkddkdd”</td>
</tr>
<tr>
<td>2</td>
<td>“dldldldldldldld”</td>
</tr>
<tr>
<td>3</td>
<td>“sdkddkddkddskdd”</td>
</tr>
<tr>
<td>4</td>
<td>“sdkddldskddskdd”</td>
</tr>
<tr>
<td>5</td>
<td>“sdkddldskddskdd”</td>
</tr>
<tr>
<td>6</td>
<td>“sdkddldskddskddskdd”</td>
</tr>
<tr>
<td>7</td>
<td>“sdkddldskddskddskdd”</td>
</tr>
<tr>
<td>8</td>
<td>“sdkddldskddskddskdd”</td>
</tr>
<tr>
<td>9</td>
<td>“sdkddldskddskddskdd”</td>
</tr>
<tr>
<td>10</td>
<td>“sdkddldskddskddskdd”</td>
</tr>
</tbody>
</table>

Table 10.8: Scheduling jobs

Finally, in Figure 10.3 the runtimes in milliseconds are given for AGENTS
and clp(FD) applied to the reprographic scheduling problem, where the in-
puts are shown in Table 10.8.

The first optimal solution is computed in each case. The AKL(FD)-
program is in average about 20% faster than the clp(FD)-program, which is
probably explained by the entailment marking optimization which is missing from clp(FD) and which is very efficient on this particular problem (Table 10.2).

In Figure 10.4 we have plotted the runtimes of clp(FD) and AGENTS from figures 10.1, 10.2, and 10.3 against each other. If a point is below the break-even line it indicates that for this run clp(FD) was faster, and vice versa.

In conclusion:

- clp(FD) is in general faster than AGENTS, ranging between being 5 times faster (the n-queens problem), to being no faster at all (the reprographic scheduling problem). This is mainly explained by the fact that clp(FD) compiles all source code to C, where as much as possible is inlined to avoid expensive procedure calls. Furthermore, the implementation is nicely fine-tuned to efficiently evaluate indexicals and perform propagation [DC93c, DC93b].

- The inclusion of FD in AGENTS is based on the generic constraint interface (Chapter 9), where no FD specifics are included in the AGENTS emulator. Indexicals are emulated by a naive stack machine, instead of compiled to C, which further degrades the performance.

- However, the performance of AGENTS is still competitive enough to claim that even a simple implementation of indexicals can clearly be the foundation on top of which powerful constraint combinators for conditional, disjunctive, and cumulative reasoning can be added.
10.2 Evaluating AGENTS with Copying

In this section we are using the copying scheme for labeling developed in Section 9.5.1.

10.2.1 Comparing with Oz

We now compare AGENTS with Oz, which offers the same basic functionality for finite domain constraints as do AGENTS [Smo95, HM95, MMP95, MPSW95].

The machine used throughout in the evaluation is a Sun 4/25 (SPARCstation ELC), running SunOS 4.1.2, and all measurements are done on this machine.

In Table 10.9 we give the timings in milliseconds for the same programs run in AGENTS and in Oz. This time, we use the copying-based backtracking of AGENTS, since this scheme is similar to the one used in Oz. Hence, we have included both the copy time (C) and the run time (R) for each system. The timings range from Oz being 3.7 times faster (eq20), which is explained by the fact that arithmetic constraints are being built into Oz as C-primitives, to AGENTS being 2 times faster (queens 64). However, copying is always more expensive in AGENTS. For the alpha problem, Oz does not terminate in reasonable time using naïve labeling, apparently due to memory allocation problems.

Furthermore, we give plotings of the two systems applied to a series of inputs to the n-queens, the magic series, and the squares programs. We
exclude the copy time in the following.

In Figure 10.5 the runtimes in milliseconds are given for AGENTS and Oz applied to the 60-, 61-, and so on up to the 69-queens problem, using first-fail labeling to compute the first solution. As can be seen, the runtimes are stochastic with respect to the number of queens. The relation remains constant between the two systems, where AGENTS is about 2.5 times as fast. However, the copying time in AGENTS is in average between 250% and 300% of the runtime where as in Oz it is only 140% in average.

In Figure 10.6 the runtimes in milliseconds are given for AGENTS and Oz applied to the magic 40-, 41, and so on up to the 49-series problem, using first-fail labeling to compute the unique solution. The execution times grow roughly linear with the size of the input for each program, and the AKL(FD)-program runs about 10% faster throughout. The copying overhead in AGENTS for this problem is in average 140% of the runtime, where as in Oz it is about 110% of the runtime.

In Figure 10.7 the runtimes in milliseconds are given for AGENTS and Oz applied to the squares problem, where the inputs are shown in Table 10.10. In the table, column Size shows the size of the master square, and column Small squares shows the list of the small squares with which to pack the master square (actually, the size of the small squares). Note that we allow multiple squares with the same size. The first solution is computed in each case, using first-fail labeling. For this problem, AGENTS is about 2 times faster in runtime, and between 2 and 3 times slower in copying. This is explained by the different treatment of disjunctions in the two systems.

In Figure 10.8 we have plotted the runtimes of Oz and AGENTS from Table 10.9 and figures 10.5, 10.6, and 10.7 against each other. If a point is below the break-even line it indicates that for this run Oz was faster, and
10.2 Evaluating AGENTS with Copying

<table>
<thead>
<tr>
<th>Program</th>
<th>AGENTS</th>
<th>Oz</th>
<th>AGENTS/Oz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R:</td>
<td>C:</td>
<td>R:</td>
</tr>
<tr>
<td>eq10</td>
<td>940</td>
<td>360</td>
<td>817</td>
</tr>
<tr>
<td>eq20</td>
<td>1540</td>
<td>620</td>
<td>416</td>
</tr>
<tr>
<td>sendmory</td>
<td>80</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>alpha</td>
<td>39380</td>
<td>14710</td>
<td>??</td>
</tr>
<tr>
<td>alpha-ff</td>
<td>460</td>
<td>570</td>
<td>306</td>
</tr>
<tr>
<td>five</td>
<td>80</td>
<td>40</td>
<td>33</td>
</tr>
<tr>
<td>sudoku</td>
<td>430</td>
<td>360</td>
<td>134</td>
</tr>
<tr>
<td>queens-ff 64</td>
<td>1810</td>
<td>5960</td>
<td>4600</td>
</tr>
<tr>
<td>magic-ff 40</td>
<td>2860</td>
<td>4120</td>
<td>3100</td>
</tr>
<tr>
<td>cars</td>
<td>170</td>
<td>160</td>
<td>100</td>
</tr>
<tr>
<td>squares 20</td>
<td>3550</td>
<td>9790</td>
<td>6200</td>
</tr>
</tbody>
</table>

Table 10.9: AGENTS versus Oz (ms.)

<table>
<thead>
<tr>
<th>Size</th>
<th>Small squares</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>[5,5,5,3,3,3,1,1,1,1,1,1,1,1,1]</td>
</tr>
<tr>
<td>12</td>
<td>[8,4,4,4,4,2,2,2]</td>
</tr>
<tr>
<td>13</td>
<td>[7,6,6,4,3,3,2,2,1,1,1,1,1]</td>
</tr>
<tr>
<td>14</td>
<td>[7,7,5,5,4,3,2,2,2,2,1,1,1]</td>
</tr>
<tr>
<td>15</td>
<td>[8,7,7,5,3,3,3,2,2,1,1,1]</td>
</tr>
<tr>
<td>16</td>
<td>[8,8,5,4,4,4,4,3,3,2,2,1,1,1]</td>
</tr>
<tr>
<td>17</td>
<td>[9,8,6,4,4,4,4,3,3,2,1,1,1]</td>
</tr>
<tr>
<td>18</td>
<td>[9,9,5,4,4,4,4,1,1,1,1,1,1,1]</td>
</tr>
<tr>
<td>19</td>
<td>[10,9,7,6,4,4,3,3,3,3,3,2,2,1,1,1,1,1]</td>
</tr>
<tr>
<td>20</td>
<td>[9,8,8,7,5,4,4,4,4,4,3,3,3,2,2,1,1]</td>
</tr>
</tbody>
</table>

Table 10.10: Square problems

vice versa.

In conclusion:

- Oz and AGENTS perform similar in average, where AGENTS is faster for problems where the FD optimizations are particularly efficient, such as the n-queens and the magic series problems, and for problems where entailment checking and constructive disjunction are crucial, such as for the squares problem. Currently, Oz does not support disjunction applied constructively, and will benefit from such. Also, Oz seems to be faster where the builtin arithmetic of Oz is used heavily, such as in the arithmetic problems in Table 10.9. It should be noted that the implementation of Oz is going through some revisions, and that the next version of the system will be faster on finite domain
constraint solving than the current version.

- Oz outperforms AGENTS in copying, where copying, many times, is more than twice as fast as in AGENTS.

- The optimized backtracking scheme, using trailing, of AGENTS is faster than the copying scheme. And, in particular, the memory consumption is much more moderate than when copying. Therefore, it is not clear that the copying scheme is appropriate to use for finite domain problems, due to the heavy backtracking that typically occurs.

10.2.2 Evaluating the disjunctive constraints

We now compare our two approaches for constructive disjunction, i.e. the indexical scheme (Section 4.2) and the constraint lifting scheme (Section 8.3) with speculative and cardinality-based disjunction (Section 3.6.5) [CC95]. As benchmarks we use two problems for scheduling and planning, the bridge-project problem [Hen89] and the squares problem (Section 2.2.1), together with the n-queens problem (Section 2.1.6). See Section 8.6 and Appendix B for the programs coding the constraints below.

The bridge and squares problems are concerned with shared resources, where the disjunctions are thus resource constraints. In the bridge example the disjunction

\[ x_1 + s_1 \leq x_2 \lor x_2 + s_2 \leq x_1 \]

is used, where \( x_1 \) and \( x_2 \) are domain variables, and \( s_1 \) and \( s_2 \) are constants.
In the squares example two disjunctions are used:

\[ x_1 + s_1 \leq x_2 \lor x_2 + s_2 \leq x_1 \lor y_1 + s_1 \leq y_2 \lor y_2 + s_2 \leq y_1 \]

where \( x_1, x_2, y_1, \) and \( y_2 \) are domain variables, and \( s_1 \) and \( s_2 \) are constants, and the disjunction

\[ (b = 1 \land x \in \{p - s + 1, \ldots, p\}) \lor (b = 0 \land x \in \{0, \ldots, p - s\} \cup \{p + 1, \ldots\}) \]

where \( x \) is a domain variable, and \( p \) and \( s \) are constants.

For the n-queens problem we consider the effect of applying lookahead pruning [Hen89] through disjunctive reasoning on the number of nondeterminate steps. Lookahead is applied by adding the disjunctive constraint member(\( i, [x_1, \ldots, x_n] \)) for each \( i \) between 1 and \( n \), where \( x_i \) represents queen \( i \) and member(\( i, [x_1, \ldots, x_k] \)) is interpreted as \( i = x_1 \lor \cdots \lor i = x_k \).

We have run the programs in AGENTS. The timings are in milliseconds computed on a SPARC-10 system. If no answer was computed within one minute, or when the memory consumption became too large, "?" is used in the tables. We have used first-fail labeling throughout.

In tables 10.11, 10.12, and 10.13 we have included the runtime and number of nondeterminate steps for planning a bridge-project with about 30 jobs and 70 constraints, for packing a square with 8 squares, and packing a square with 17 squares, using speculative disjunction (spec), cardinality-based disjunction (card), and disjunctions applied constructively, either using the indexical coding (weak) or constraint lifting (strong).
As can be seen, weak and strong disjunction prunes the number of non-determinate steps more than do speculative and cardinality-based disjunction, however, the indexical scheme outperforms lifting in runtime. This is because constraint lifting does not produce sufficiently more pruning than the indexicals do, while being much more expensive in time and space.

As we see it, the most problematic aspect of applying constraint lifting is the reactivity of the disjunction. Each disjunct may affect, or be constrained by, many other variables. Hence, for any update of any one of these variables in the embedding store, the disjuncts must be reconsidered, and lifting retried. This should be controlled somehow, e.g. by only applying lifting at certain stages in the computation, and not necessarily at each propagation.

The reason why speculative disjunction needs fewer nondeterminate
steps than cardinality in the bridge-example is that the solution happens to be found early in the speculative search. However, the execution of the program using speculative disjunction is heavily burdened by expensive deep guard propagations in AGENTS.

In Table 10.14 we give the timings for running the n-queens program with the extra member/2 disjunctions added and their four different interpretations, together with the version of n-queens with no extra disjunctions added (no).

As seen from the table, the member/2 constraint lifting prunes the number of nondeterminate steps dramatically, however, at a large performance cost. The speculative, weak and cardinality-based disjunctions prune the number of nondeterminate steps somewhat, however, with no obvious performance gain. We have also experimented with adding the redundant constraint \( x_i = 1 \lor \cdots \lor x_i = n \), for each \( x_i \), which for this example was less efficient than the member/2 constraint.
In conclusion; applying disjunctions constructively pays off. The indexical scheme performs surprisingly well, pruning quite heavily in the above scheduling and packing examples at a low cost. Constraint lifting on the other hand is powerful but hard to control. Some strategies need to be defined which will better exploit the pruning capacity of lifting without naively applying it whenever possible. This is analogous to how nondeterminism is controlled using stability.
Chapter 11

Related Work

We give an overview of programming systems and languages that have had seminal impact on constraint programming with finite domains, and that have influenced our work on AKL(FD). However, first we give a brief background of the origins of finite domain constraints and constraint programming.

11.1 Constraint Satisfaction Problems

Constraints for problem solving originates with the work on symbolic logic and mathematical programming in the 50s and 60s [NS56, McC60, DP60, Dan63, Rob63, Min67], ideas which were further developed by the symbolic and integer programming communities [Hew69, CKPR73, Kow74, BM76, WP77, SS80, PS82, Col84, JL87, DHS88] (for good overviews see [Coh90], [Hen91], and [JM94]), and by work on constraint solving algorithms in algebra and operations research [Dan63, Bak74, GJ79, PS82].

The key idea is to specify a set $C$ of constraints, which are first-order open formulas, such that the extension of $C$ contains precisely the solutions to a given problem, hence the naming “constraint satisfaction” problems [Mac88]. However, a very large class of problems can be described by constraints, including the class of NP-complete problems, and thereby there is no efficient method for solving all constraint satisfaction problems.

By using logical inference systems, typically based on resolution or some other proof system, and combining these with algebraic solver algorithms, efficient methods have been developed for many classes of constraint problems (for an overview see [JM94]). As an archetypal class consider the class of scheduling problems.
11.1.1 Scheduling problems

Scheduling problems were early recognized as important and computationally challenging [MT63, Bak74, Kan76, GJ79, AC91, Got93, AB93]. They occur everywhere, since timetables and production plans are a vital part of most workplaces. Furthermore, resource allocation problems constitute a subclass of the scheduling problems, and these are fundamental in industrial design and production. Scheduling problems characteristically contain constraints of the form:

- **preference** $s_i + d_i \leq s_j$, stating that some task $i$ must be completed before another task $j$ begins.

- **exclusion** $s_i + d_i \leq s_j \lor s_j + d_j \leq s_i$, forcing two tasks, $i$ and $j$, which share a resource, to avoid using the resource simultaneously.

- **consumption** $\sum r_i \leq l_j$, stating that the total consumption of resources at point $j$ does not exceed the limit $l_j$, where $r_i$ equals the amount of resources consumed by task $i$ at point $j$.

The operations research community has been involved in designing algorithms for scheduling and allocation for more than 30 years. Their work has also led to a classification of subclasses of well-analyzed scheduling problems for which quite efficient average-case algorithms exist:

- **project-planning** i.e. problems with only precedence constraints, and typically with a large amount of tasks.

- **flow-shop** i.e. problems with precedence constraints between tasks contained in the same job, and each job requires the same resources in the same order as the other jobs do.

- **job-shop** i.e. problems with precedence constraints between tasks contained in the same job, and where the sequence of resource requirements between jobs may vary.

- **open-shop** i.e. problems with no precedence constraints and with arbitrary resource requirements.

Today much work is devoted to adapting the scheduling algorithms for constraint programming such that they can be used incrementally and flexibly, and also mixed with other types of solver algorithms [AB93, BC94, ILO93, Cas91, CGL93, Pug94, CL94].

11.2 Constraint Programming Systems

We will now look more into detail of those programming systems which are most closely related to AKL(FD) and which have influenced our work considerably. We focus on the FD-part, since AKL has already been thoroughly described and analyzed elsewhere [Jan94].
11.2 Constraint Programming Systems

11.2.1 Forerunners

There are four notable forerunners of concurrent finite domain constraint programming: Prolog, Alice, GHC, and Prolog-III.

Prolog was the first efficient programming language based on resolution [CKPR73, WP77], and has since its invention been extended and refined into being a versatile and powerful programming language [AAB+93]. Many of the features of Prolog have also been adapted to concurrent logic programming languages, in particular, the symbol manipulation, which is very well suited for manipulating formal languages and data structures.

Prolog-III (and Prolog-II) was a sequel to Prolog [Col90], and was seminal in its approach to constraint solving within Prolog. Its constraint system treats rational trees, lists, numbers and booleans, and showed the way for much research to come.

GHC is a concurrent logic programming language which introduced a clean model for guards, suspensions, and commit [Ued85], which was brought over into the design of AKL [JH91]. Today, however, the developments of the systems have diverged, where the GHC research is directed towards the flat part of the language, and the AKL research concerns more interoperability, distribution, and parallelism.

Alice was a powerful constraint solving system which introduced many new constraints and combinators [Lau78], some which are still neither fully understood nor widely used in the constraint programming community. For example, constraints over finite functions and sets were allowed, so that one could constrain a finite function to be bijective simply by stating bij($f$).

We believe that Alice struck a chord which still has much to offer modern constraint programming languages.

11.2.2 CCP

The introduction of the concurrent constraint programming (CCP) languages showed how constraint solving could be used not only for problem solving, but also for communication and synchronization between processes [Mah87, Sar89, Sar93]. The basic insights are:

- A set of constraints can be considered to be a store, corresponding to the memory state of a computation. By associating each store with its set of solutions, stores can be partially ordered, and programs can thereby be denoted by closure operators over stores [SRP91].

- Synchronization is handled by entailment checking, i.e. a computation can be made to suspend (block) until a certain constraint is entailed by the store, i.e. until each solution of the store is a solution of the constraint. Entailment checking as used in CCP languages also provides a good mechanism for control in user-defined constraints [HSD91, CJH94, MPSW95].
• Communication is handled by adding (telling) constraints to the store, which increases the amount of information in the store. Thus, by making two processes share variables, one process can suspend through a constraint on a variable \( x \) say, until the other process adds a sufficiently strong constraint on \( x \). A simple case is a list of items, where one process, the consumer, suspends until the head of the list contains an element, and the other process, the producer, adds successive constraints on the list.

• Information can be removed only by backtracking. This simplifies concurrent programming, since the order of store updates is not crucial. For example, in an imperative language, the order of execution of the statements \( x := 1 \) and \( x := 2 \) makes all the difference when \( x \) is a variable shared between two processes. However, the two constraints \( x = 1 \) and \( x = 2 \) are incompatible, and therefore such concurrency traps are avoided in a CCP language. Instead, CCP programs are structured around partial information which monotonically increases, which gives the programs a natural denotational semantics.

AKL is a deep-guard CCP language, and its design has been deeply affected by the groundbreaking work in [Sar89].

11.2.3 CHIP

CHIP (Constraint Handling In Prolog) is a constraint logic programming language, originating at ECRC, and further developed at Cosytec [DHS+88, AB91, AB93, BC94]. The language contains solvers for constraints over

• finite domains,

• booleans, and

• rational numbers.

We only address the finite domain part herein.

The syntax of CHIP is that of Prolog, extended with operators for finite domain equalities, inequalities, and disequalities. A number of builtin symbolic constraints are provided, including element/3, atmost/3, and atleast/3. As a parenthesis, the notion of “symbolic” constraints seems to come from CHIP, where it is used to denote non-arithmetic constraints over finite domain variables. However, as is shown in Chapter 3, these beforementioned symbolic constraints really are nothing but constraints in FDC, i.e. propositional expressions over finite domain arithmetic constraints.

Furthermore, user-defined constraints are introduced in CHIP, using global declarations. Such constraints are declared as being forward-driven or lookahead-driven, thus stating suspension conditions on the constraints [Hen89].
Recently, cumulative and global constraints were added to CHIP to efficiently deal with scheduling and packing problems [AB93, BC94].

Finally, some primitives are provided for finding solutions to a list of constrained variables, e.g. labeling/1, labelingff/1, minimize/2, and minimize_maximize/2.

In conclusion;

- CHIP has been, and continues to be, very influential. Partly by being efficient and powerful, and partly by being innovative. However, there are some disadvantages.

- First of all, the mechanisms for user-defined are not sufficient. The programmer needs more control and power than what is provided by the declarations. Many constraints in CHIP simply act as a wrapper of a huge chunk of C-code, cumulative/4 being perhaps the topmost example, where it is claimed that almost 20 000 lines of C-code are used to implement the constraint. Preferably, most of the power of the cumulative constraints could be captured at a higher-level if the constraint programming language provided the right set of primitives.

- Secondly, the programmer needs more control over the labeling procedures. Search-heuristics differ between different classes of problems and it is unlikely that a fixed set of heuristic primitives will suffice for all classes.

### 11.2.4 cc(FD)

The language cc(FD) was proposed as a remedy to some of the above mentioned problems in CHIP [HSD91, HSD92a]. In particular, the language provides better primitives for programming user-defined constraints, and is based on the CCP paradigm.

As basic building blocks there are the arithmetic constraints. A distinction is made between interval-reasoning and domain-reasoning arithmetic constraints, and different operators are provided for respective case.

On top of these constraints, new (symbolic) constraints are defined using three combinators:

- **cardinality**  
  cardinality expressions are used for adding constraints to the store in a controlled and reactive fashion, i.e. by keeping track of the number of true (false) constraints in a list, when thresholds are passed, the cardinality operator reacts correspondingly (Section 3.6.4).

- **implication**  
  blocking implication is a paraphrase of the ask-combinator of CCP, and blocks a computation until a certain constraint is true (entailed).
disjunction. Constructive disjunction was here shown to be a powerful new construct with nice model-theoretic semantics. However, very little has been revealed of its implementation, and its relation to the disjunctions proposed earlier (Section 3.6.5) is unclear.

cc(FD) has been the number one influence on our work. Here indexicals were first introduced [HSD91], as well as constructive disjunction and blocking implication. The use of antimonotone indexicals for entailment checking was mentioned without being fully worked out, and the basic aspects of monotonicity of indexicals were explained. We see our work as picked up from the point where the indexicals were defined, and henceforth we have developed efficient compilation and execution algorithms for the indexicals, as well as integrating them in a deep-guard CCP language.

11.2.5 clp(FD)

Clp(FD) is a constraint logic programming (CLP) language in which the indexicals have also been integrated [DC93c, DC93a, DC93b, Dia94]. This system is designed to be highly efficient, where programs are compiled to C-code containing a minimal number of procedure calls. The indexicals are compiled to abstract machine instructions, fine-tuned to optimize propagation of domain constraints.

The syntax of clp(FD) resembles the syntax of CHIP, which is its closest peer. User-defined constraints are either provided as clp(FD) programs, or as a combination of clp(FD) programs and C-functions designed using guidelines [DC93b, Dia94]. The system is built on top of the GNU C-compiler (gcc), and generates Unix-executables when compiling a source program.

They basically provide the same indexical language as we do, with the exception of the conditional ranges. Their solver is similar to ours, but exploits slightly different optimizations [DC93c]. The system provides some primitives for meta-programming, such that certain heuristics can be programmed at source level.

The arithmetic constraints in clp(FD) include both linear and nonlinear constraints. A few symbolic constraints are included, such as element/3 and atmost/3, as well as primitives for branch-and-bound (minof/2 and maxof/2). Furthermore, a package of boolean constraints is predefined, which is based on an efficient specialization of FD for boolean numbers [DC93a].

In conclusion, clp(FD) is fast, and shows that user-defined constraints in terms of indexicals can in fact be made as efficient as builtin handling of constraints can be. The language currently does not offer all the necessary primitives for constraint programming, such as constructive disjunction and cumulative constraints; however, the basic foundation has been laid upon which such constructs should fit nicely.
11.2.6 Oz

Oz is a concurrent language allowing functional, object-oriented and constraint programming, based on the CCP paradigm [Smo95, HM95, MMP95, MPSW95]. Similar to AKL, Oz provides deep guards, and the same three guard operators for conditional, committed and nondeterminate choice are included. The language is higher-order which enables a theoretically appealing treatment of objects and inheritance [Smo94].

Oz includes a combinator for encapsulated search [SS94], which makes many search strategies, including depth-first, breadth-first, and branch & bound to be programmable at source level. Furthermore, by providing primitives for accessing information related to finite domain variables, heuristics such as first-fail can also be programmed at Oz level [MPSW95].

The finite domain constraints are integrated in Oz through a set of C-builtins for arithmetic and symbolic reasoning. The execution of these builtins is controlled using guarded clauses, whereby some disjunctive and conditional reasoning can be performed [MPSW95]. Some differences with respect to AKL(FD) are:

- Arc-consistency is maintained by constraints programmed in C, so called propagators, instead of using indexicals.
- There is no constraint lifting operator, hence, neither weak nor strong constructive disjunction can be programmed.
- No trailing scheme is provided for backtracking, instead copying is used.

11.2.7 ILOG SOLVER

ILOG SOLVER (or SOLVER for short) is a C++ library which embodies the CLP paradigm, i.e. the programmer is provided constraint variables, constraints, and stores [IL093, Pug94]. Propagation is performed similar to CLP/CCP systems by applying arc-consistency, and mechanisms for search and trailing are included in the library.

In SOLVER, a constraint variable is a C++ object. There are four basic classes:

- integers, with an associated finite domain or interval,
- floating point intervals,
- booleans, and
- finite set intervals.
A constraint variable is thus treated as any object is in C++, meaning that the responsibility of allocation and deallocation is handed over to the programmer. However, SOLVER does provide a way of constructing variables such that space is reclaimed upon backtracking.

The library comes with the standard support for arithmetic and symbolic constraints, where the operator overloading of C++ is heavily used for syntactic sugaring. For example, the C-expression \( x \equiv y + z \) can be made to be equivalent to \( x = y + z \), where \( x \), \( y \), and \( z \) are constrained variables.

It is worth noting that SOLVER also provides a mechanism for user-defined constraints such that they can be exploited by the arc-consistency algorithm. Basically, the new constraint inherits from the basic constraint class, and extends the virtual functions (methods) of the base class. It is claimed that this is crucial to the success of a constraint programming language [Pug94].

Furthermore, the programmer has good control over search and labeling. First of all, a generic labeling procedure is defined, which can be provided a function for selecting the variable to enumerate next, i.e. a heuristic such as first-fail can be programmed. Secondly, when instantiating variables, which value to select next can also be controlled, such that value-based heuristics can be implemented. Thirdly, nondeterministic C++ procedures can be defined, using macros for defining clauses, such that search paths can be selected as wanted, using the fact that the solver searches depth-first.

SOLVER is a solid base for efficient and versatile constraint programming, as is shown for example by the powerful system for scheduling problems developed on top of SOLVER [Pap04]. SOLVER is imperative, thereby, constraint programming is more easily integrated in real applications; however, by paying the prize of imperative programming, i.e. memory management becomes a major concern, and programming becomes more error-prone than when using a symbolic language.
Chapter 12

Conclusion

We summarize our contributions, and describe the work we are currently engaged in.

12.1 Thesis Contribution

Our thesis develops and extends the notion of indexicals, and relates it to a language FDC of finite domain constraints. A constraint in FDC is an arithmetic constraint, or a propositional combination thereof. We show how the indexicals alone can be used for encoding the consistency and entailment of FDC such that:

- partial or full arc-consistency of the arithmetic constraints can be checked by a solver for indexicals,
- implication can be blocking,
- a weak, but sufficient, kind of constructive disjunction is realized, and that
- entailment of any constraint in FDC can be checked by the entailment checking algorithm for indexicals.

The solver for indexicals is defined, and its complexity is shown to be the same as for the traditional arc-consistency algorithm. The algorithm treats monotone and antimonotone indexicals similarly such that it can check both arc-consistency and entailment without sacrificing performance. We describe a simple, but sufficient, implementation of the solver, and clarify important optimizations. This includes compiler algorithms for extracting information used for optimizing propagation, and for computing monotonicity.

The second part of the thesis is concerned with integrating the indexical system in AKL, a deep-guard concurrent constraint language. This adds
complications. Since AKL allows deep guards, the solver for indexicals no longer executes in a single store, but in a hierarchy of stores. This implies that the indexicals are structured by a hierarchy too, and that suspensions between variables and indexicals can cross-refer between stores. The solver must then propagate between stores, as well as within one store.

The integration was done using a generic constraint interface, such that no adjustments were made to the emulator of AKL, but instead methods, provided by the interface, were used through which entailment checking, the hierarchy maintenance, and nondeterminate search are handled.

The compiler of AKL was extended such that indexicals and FDC constraints can be compiled either for arc-consistency, or for entailment checking. Thereby, the guard combinators of AKL can be used for defining powerful finite domain constraints as AKL-programs.

Furthermore, we added another guard operator to AKL which performs constraint lifting, a mechanism by which constructive disjunction of finite domain constraints can be implemented. We evaluate this operator, and compare it with other kinds of disjunction, including the one proposed in the translation of disjunctive FDC constraints to indexicals.

Finally, we evaluate the implementation of AKL(FD), AGENTS, and compare it with other similar languages. It is shown that AGENTS is competitive, and that it offers new programming techniques.

12.2 Future Work

We have several ongoing activities, all concerned with extending our thesis work to be more powerful and applicable. We are considering three objectives.

First, arc-consistency of arithmetic and disjunctive constraints alone does not suffice when attacking truly hard scheduling and verification problems. Adding clever enumeration procedures helps, however, the constraint solvers must be more powerful too. Using our current work on domain propagations, we see a large potential in also allowing more general constraint propagation, i.e. not only should variable prunings be propagated, but also truth assignments to constraints. There should be no conceptual nor operational difference between assigning a constraint to be true/false from assigning a variable some values. This has previously been shown to be extremely effective for propositional logic [Stä94], and by combining our work on FDC, with such propagation for propositional expressions, a more powerful propagation algorithm is achieved.

Secondly, a constraint programming language should allow the programmer to analyze the problem at hand. For example, given a graph coloring problem, some clique in the graph should be colored beforehand, to eliminate useless search. Hence, a programmer must be able to compute such information, which is not possible in general in current constraint program-
ming languages. Arguably this is as important as providing good builtins for search and enumeration.

Thirdly, the use of finite domain constraint programming should spread outside the logic programming community. The techniques are powerful, and touch areas which are part of almost all industrial production, therefore, much more use could be made of finite domain constraints than what is the case. However, today most finite domain applications are written inside CLP-languages, or using licensed C-libraries, which slows the acceptance of constraint programming in the larger context.

We address these objectives as follows:

1. Building on previous work on efficient propagation algorithms for propositional logic, we are currently extending these algorithms with support for finite domain constraints, where our thesis work is directly applicable. This work will produce new constraint solvers, which extend arc-consistency in a non-trivial way.

2. Finite domain constraints are being integrated in SICStus Prolog, again using our thesis work. Consequently, the system will be extended with better support for analysis of constraint graphs, dynamic generation of constraints, and intelligent backtracking. Also, techniques developed under item 1 will be reused in here.

3. Erlang is a concurrent functional language, developed at Ellemtel Laboratories [AVW93]. It has a natural notion of processes and distribution, which makes the language perfect for building large, reactive and distributed applications. We have initiated a work on adding FD to Erlang, thus making the language applicable to a new set of problems, nicely complementing its concurrency and distribution features. Erlang(FD) will thereby contain the necessary notions for developing large-scale control and supervision systems, reactive scheduling and verification applications, intelligent agents, and interactive planning tools, which are exciting new domains of software.

4. Disjunctive and cumulative constraints need more attention. The work on cumulative constraints and scheduling libraries [AB93, BC94, Pap94], which constitute state-of-the-art of constraint satisfaction applied to scheduling and planning, should be complemented with fundamental investigations of how a propositional language of finite domain constraints, such as FDC, combined with the techniques addressed in item 1 and 2, can reach similar performance figures.

5. Theoretical investigations of the complexity bounds of propagation algorithms (arc-consistency and similar) applied to specific problem classes, such as scheduling problems, are needed to better understand the limitations of constraint propagation. In addition, we intend to
study the use of surrogate (redundant) constraints to improve upper and lower bounds for such problem classes.

Hence, the research on finite domain constraints has come far, but still many problems must be solved before the techniques are as general, robust, and powerful as required.


Appendix A

AKL(FD) libraries

We now give example definitions of some important library constraints in AKL(FD).

A.1 Arithmetic builtins

The following definitions are written such that the peephole optimizations in Table 6.7 apply. We use x//y to denote \(|x/y|\).

The constraint \(c \times x = y\) is defined as:

\[
'c\times y'(C,X,Y) :- \\
f( \\
    Y \text{ in } C \times \text{min}(X) .. C \times \text{max}(X), \\
    X \text{ in } \text{min}(Y) // C .. \text{max}(Y) / C ) . \\
\]

The constraint \(x + y = z\) is defined as:

\[
'x+y=z'(X,Y,Z) :- \\
f( \\
    Z \text{ in } \text{min}(X) + \text{min}(Y) .. \text{max}(X) + \text{max}(Y), \\
    Y \text{ in } \text{min}(Z) - \text{max}(X) .. \text{max}(Z) - \text{min}(X), \\
    X \text{ in } \text{min}(Z) - \text{max}(Y) .. \text{max}(Z) - \text{min}(Y) ) . \\
\]

The constraint \(x + c = y\) is defined as:

\[
'x+c=y'(X,C,Y) :- \\
\text{Cn is } -C , \\
f( \\
    X \text{ in } \text{min}(Y) + C .. \text{max}(Y) + C, \\
    Y \text{ in } \text{min}(X) + \text{Cn} .. \text{max}(X) + \text{Cn} ) . \\
\]

The constraint \(x + c \neq y\) is defined as:
A.2 Symbolic builtins

The following definitions are written to exploit the special range functions in Section 6.6.2.

The constraint element($i, l, x$) is optimized by using the element_x and element_i instructions in Section 6.6.2 and is defined as:

\[
\text{element} (I, L, X) :-
\text{integer_list}(L)
\text{-> fd( I in 1.. ),}
\text{fd( X in element_x(I,L),}
\text{I in element_i(X,L) ).}
\]

\[
\text{integer_list}([], X) :-
\]
\text{-} \text{ true.}
\text{integer_list([I|R]) :-}
  \text{ integer(I)}
  \text{ \text{-} integer_list(R).}

The constraints \text{atmost}(n,l,v) and \text{atleast}(n,l,v) are defined as:

\text{atleast}(N, L, V) :-
  \text{-} \text{ fd(M in N.. ),}
  \text{ count(M, L, V).}

\text{atmost}(N, L, V) :-
  \text{-} \text{ fd(M in 0..N ),}
  \text{ count(M, L, V).}

where \text{count}(n,l,v) is defined as:

\text{count}(N, [], _) :-
  \text{-} \text{ fd( N in 0 ).}
\text{count}(N, [X|L], V) :-
  \text{-} \text{ eq_iff(X, V, B),}
  \text{ fd( B+M=N ),}
  \text{ count(M, L, V).}

\text{eq_iff}(X, V, B) :-
  \text{ fd(B in 0..1 ),}
  \text{ fd(X in \text{dom}(B)/\%1) \Rightarrow V \ \backslash \ \text{dom}(B)/\%0) \Rightarrow (-V),}
  \text{ B in \text{dom}(X)/\%V) \Rightarrow 1 \ \backslash \ \text{dom}(X)/\%0) \Rightarrow 0 ).
Appendix B

AKL(FD) programs

We give a listing of the corresponding AKL(FD) programs for the example problems in Section 2. Some parts of the programs are left out which are not connected with constraint programming but with bookkeeping and data manipulation.

B.1 10 and 20 Equations

For a description of the problems see Section 2.1.1 and 2.1.2. For this problem no particular enumeration is needed, the variables can be assigned in an arbitrary order:

```plaintext
eq 10 (Lab, LD):-
    LD = [X1, X2, X3, X4, X5, X6, X7],
    domain(LD, 0, 10),
    fd(LD)
```

```
  0 + 98627 * X1 + 45883 * X2 + 872 * X3 + 89422 * X5 + 66159 * X7
  1 = 54704 + 30704 * X4 + 50649 * X6,
  0 + 6867 * X2 + 9863 * X3 + 69966 * X4 + 62038 * X5 + 57164 * X6 + 38413 * X7
  2 = 18238 * 3 + 9389 * X1,
  90003 * 2 + 106349 * X1 + 77761 * X2 + 6702 * X5
  3 + 609197 * X3 + 61944 * X4 + 92964 * X6 + 44660 * X7
  5 = 73947 * 1 + 84391 * X3 + 51310 * X5
  1164383 * X2 + 344247 * X4 + 70882 * X6 + 33604 * X7
  0 + 13067 * X3 + 42230 * X4 + 77627 * X5 + 96662 * X7
  1186471 * 1 + 60162 * X1 + 21103 * X2 + 97932 * X6
  1394152 * 2 + 66920 * X1 + 83679 * X4
  0 + 642034 * X2 + 166377 * X3 + 46881 * X5 + 67707 * X6 + 98038 * X7
  0 + 688860 * X1 + 27886 * X2 + 31716 * X3 + 73697 * X4 + 388336 * X7
  279091 * 1 + 69863 * X5 + 76391 * X8
  0 + 76132 * X2 + 71760 * X3 + 22770 * X4 + 65211 * X5 + 78587 * X6
  469023 * 1 + 82817 * X7,
  519878 * 94198 * X2 + 677234 * X3 + 37408 * X4
  0 + 71563 * X1 + 26728 * X6 + 26495 * X6 + 70023 * X7,
  361921 * 79693 * X1 + 325592 * X5 + 38478 * X6
  0 + 94129 * X2 + 43188 * X3 + 82858 * X4 + 69028 * X7
```
eq20(Lab,LD):-
LD - [X1,X2,X3,X4,X5,X6,X7],
domain(LD,0,10),
fld(876370+16105*X1+6704*X3+66610*X6
- 0+62397*X2+43430*X4+56100*X5+65301*X7,
63909+68722*X3
- 0+61673*X1+67761*X2+96861*X3+3634*X4+59190*X5+15280*X7,
916653+34121*X2+33488*X7
- 0+1671*X1+10763*X3+80609*X4+42632*X5+93620*X6,
129768+11119*X2+38876*X4+14413*X5+29234*X6
- 0+71202*X1+73017*X3+72370*X7,
752447+68412*X2
- 0+6874*X1+73947*X3+17147*X4+62236*X5+16006*X6+5632*X7,
90614+18810*X3+5219*X4+79785*X7
- 0+68368*X1+64120*X2+6013*X5+78189*X6,
1388290+60366*X1+4578*X3
- 0+61830*X2+96120*X4+21231*X5+49719*X6+66661*X7,
184665+64919*X1+69024*X4+76542*X5+47935*X7
- 0+60460*X2+90840*X3+25145*X6,
0+43525*X2+92298*X3+68630*X4+92690*X5
- 1503688+43277*X1+9372*X6+60227*X7,
0+47385*X2+97715*X3+69028*X5+76212*X6
- 1294867+19836*X1+12640*X4+81102*X7,
0+31227*X2+9386*X1+73889*X4+81526*X5+68026*X7
- 1410723+60301*X1+72702*X6,
0+94016*X1+30961*X3+66697*X4
- 26334+92071*X2+30706*X5+44404*X6+38304*X7,
0+94760*X2+21239*X4+81675*X5
- 277271+67466*X1+61653*X3+93936*X6+24654*X7,
0+29968*X2+87308*X3+87899*X4+4667*X6+34339*X7
- 246912+89698*X1+78219*X5,
0+86176*X1+57900*X4+18583*X5+60647*X6+63287*X7
- 373864+65332*X2+12668*X3,
0+47756*X2+19346*X4+70072*X5+44420*X7
- 740061+10343*X1+11782*X3+36991*X6,
0+49149*X1+52871*X2+66728*X4
- 146074+7132*X3+33676*X5+49530*X6+62069*X7,
0+29475*X2+34421*X3+62646*X5+29278*X6
- 261591+60113*X1+76870*X4+16212*X7,
22167+29101*X2+6613*X3+23219*X4
- 0+67069*X1+22128*X2+7276*X6+57308*X7,
921225+76706*X1+85614*X6+41906*X7
- 0+98205*X2+23446*X3+67921*X4+24111*X5,
},
enum(Lab,LD).

B.2 SEND+MORE=MONEY

For a description of the problem see Section 2.1.3. For this problem no particular enumeration is needed, the variables can be assigned in an arbitrary
B.3 Alpha

For a description of the problem see Section 2.1.4. For this problem first-fail labeling is preferred.

```
alpha(Lab,LD):-
    domain(LD,1,26),
    all_different(LD),
    fd(
        B + A + L + L + E + T = 45,
        C + E + L + L + 0 = 43,
        C + O + N + C + E + R + T = 74,
        F + L + U + T + E = 30,
        F + U + G + U + E = 50,
        G + L + E + E = 66,
        J + A + Z + Z = 58,
        L + Y + R + E = 47,
        O + B + O + E = 53,
        O + P + E + R + A = 65,
        P + 0 + L + X + A = 59,
        Q + U + A + R + T + E + T = 60,
        S + A + X + O + P + H + 0 + N + E = 134,
        S + C + A + L + E = 51,
        S + O + L + 0 = 37,
        S + O + N + G = 61,
        S + O + P + R + A + N + 0 = 82,
        T + H + E + M + E = 72,
        V + I + O + L + I + N = 100,
        W + A + L + T + Z = 34
    ),
    enum(Lab,LD).
```

B.4 Five Houses

For a description of the problem see Section 2.1.5. For this problem no particular enumeration is needed, the variables can be assigned in an arbitrary order.

```
five(Lab,L):-
    L=[M1,M2,M3,M4,M5,
        C1,C2,C3,C4,C5],
```
P1,P2,P3,P4,P5,
A1,A2,A3,A4,A5,
D1,D2,D3,D4,D5).
domain([1,1,5]),
N5=1, D5=3.
all_different([C1,C2,C3,C4,C5]),
all_different([P1,P2,P3,P4,P5]),
all_different([N1,N2,N3,N4,N5]),
all_different([A1,A2,A3,A4,A5]),
all_different([D1,D2,D3,D4,D5]).
fd(
  N1 = C2, N2 = A1, N3 = P1, N4 = D3,
  P3 = D1, C1 = D4, P6 = A4, P2 = C3, C1 = C5+1
),
p_or_m(A3,P4,1),
p_or_m(A5,P2,1),
p_or_m(N5,P4,1),
enum((Lab,[C1,C2,C3,C4,C5,
P1,P2,P3,P4,P5,
N1,N2,N3,N4,N5,
A1,A2,A3,A4,A5,
D1,D2,D3,D4,D5]).

p_or_m(X,Y,Z) :-
  fd( X in (sin(Y)+min(Z) .. max(Y)+max(Z)) \/
       (sin(Y)-min(Z) .. max(Y)-min(Z)),
       Y in (sin(X)+min(Z) .. max(X)+max(Z)) \/
       (sin(X)-min(Z) .. max(X)-min(Z)),
       Z in (sin(Y)-max(X) .. max(Y)-min(X)) \/
       (sin(Y)-min(X) .. max(Y)-min(Y) )).

B.5 N-queens

For a description of the problem see Section 2.1.6. For this problem first-fail labeling is preferred.

q_c(D,E,S) :-
  fd( D in -dom(E) \/(dom(E)+S) \/(dom(E)-S)),
  E in -dom(D) \/(dom(D)+S) \/(dom(D)-S)).

queens(lab, N, L) :-
  get_domains(N, N, L),
  constrain(L),
  enum(lab, L).

get_domains(0, _, L) :-
  -> L=[].
get_domains(N, N, L) :-
  -> L = [D|L0],
  fd( D in 1..N ),
  M1 = M-1,
  get_domains(M1, N, L0).

constrain([]) :-
B.6 Sudoku

For a description of the problem see Section 2.1.7. For this problem no particular enumeration is needed, the variables can be assigned in an arbitrary order.

```prolog
sudoku(Label, Problem) :-
  problem(Problem),
  domain_problem(Problem),
  row_constraint(Problem),
  column_constraint(Problem),
  block_constraint(Problem),
  enum_problem(Problem, Label).

domain_problem([]).
domain_problem([P|R]) :-
  domain(P, 1, 9),
  domain_problem(R).

row_constraint([R|Rt]) :-
  all_different(R),
  row_constraint(Rt).
row_constraint([]).

column_constraint([C1,C2,C3,C4,C5,C6,C7,C8,C9]) :-
  column_constraint(C1,C2,C3,C4,C5,C6,C7,C8,C9).

column_constraint([C1,C1t], [C2,C2t], [C3,C3t], [C4,C4t],
  [C5,C5t], [C6,C6t], [C7,C7t], [C8,C8t], [C9,C9t]) :-
  all_different([C1,C2,C3,C4,C5,C6,C7,C8,C9]),
  column_constraint(C1t,C2t,C3t,C4t,C5t,C6t,C7t,C8t,C9t).
column_constraint([],[],[],[],[],[],[],[],[]).

block_constraint([C1,C2,C3,C4,C5,C6,C7,C8,C9]) :-
  block_constraint(C1,C2,C3),
  block_constraint(C4,C5,C6),
  block_constraint(C7,C8,C9).

block_constraint([C1,C2,C3,C1t], [C4,C5,C6,C2t], [C7,C8,C9,C3t]) :-
  all_different([C1,C2,C3,C4,C5,C6,C7,C8,C9]),
  block_constraint(C1t,C2t,C3t).
block_constraint([],[],[]).
```

```prolog
-> true.
constraint([D|R]) :-
  -> constraint_each(R, D, 1),
    constraint(R).

constraint_each([..]) :-
  -> true.
constraint_each([E|R], D, S) :-
  q_c(D,E,S).
  S1 = S+1,
  constraint_each(R, D, S1).
```
enum_problem(P, L) :-
    append_all(P, Pf),
    enum(L, Pf).

problem(P) :-
P= [[8, _, _, _, _],
    [_, 1, 2, 3, _, _],
    [_, 4, 5, 6, _, _],
    [_, 7, 8, _, _, _],
    [9, _, _, _, _, _],
    [2, _, _, 6, 5, _],
    [_, 4, _, 3, 2, 1],
    [_, _, _, _, _, 9]].

B.7 Magic series

For a description of the problem see Section 2.1.8. For this problem no particular enumeration is crucial, however, choosing the variables in ascending order of their domain sizes respectively (the first-fail principle [Hen89]) does give a small improvement over an arbitrary ordering.

magic(Lab, N, L):-
get_list(N, L),
    N1 is N-1,
    domain(L,0,N1),
    constraints(L,L,0,N,N),
    enum(Lab,L).

get_list(0, L) :-
    L=[].

get_list(N, L) :-
    L = [[L0]],
    N1 is N-1,
    get_list(N1, L0).

constraints([], _, S0, S1) :-
    S0=0,
    S1=0.

constraints([X|Xs], L, I, S2) :-
    num(L, I, X),
    I1 is I+1,
    fd( S1+X=S ),
    c_0(I, X, S2, S3),
    constraints(Xs, L, I1, S1, S3).

c_0(0, _, S0, S1) :-
    S0=S1.

c_0(I, X, S0, S1) :-
    fd( I+X=S1=S0 ).

sum([], S) :-
    fd( S=0 ).
B.8 Squares

For a description of the problem see Section 2.2.1. In this program we use indexicals for the conditional reasoning. For this example the ordering in which the search is conducted is crucial. At each step in the search, care should be taken to place a variable at the smallest available position [Hen92].

```prolog
sum([X|Xs], I, S) :-
    -> sum(Xs, I, S1),
        eq_iff(X, I, B),
        fd(X =+ S + S1).

squares(Label, D, Xs, Ys) :-
    generate_squares(Xs, Ys, Sizes, Size),
    state_no_overlap(Xs, Ys, Sizes, D),
    state_capacity(Xs, Ys, Sizes, D),
    state_capacity(Ys, Sizes, D),
    enum(Label, Xs),
    enum(Label, Ys).

generate_squares(Xs, Ys, Sizes, Size) :-
    fd_squares_data.size_master(Size),
    generate_coordinates(Xs, Ys, Sizes).  

state_no_overlap([], [], []).  
state_no_overlap([X|Xs], [Y|Ys], [S|Ss], D) :-
    no_overlap(D, X, Y, S, Xs, Ys, Ss),
    state_no_overlap(Xs, Ys, Ss, D).

state_no_overlap(_, _, _, []).  
state_no_overlap(X, Y, S, [X|Xs], [Y|Ys], [S|Ss], D) :-
    no_overlap(D, X, Y, S, Xs, Ys, Ss),
    state_no_overlap(Xs, Ys, Ss, D).

no_overlap(spec, X1, Y1, S1, X2, Y2, S2) :-
    -> no_overlap_spec(X1, Y1, S1, X2, Y2, S2).
no_overlap(card, X1, Y1, S1, X2, Y2, S2) :-
    -> no_overlap_card(X1, Y1, S1, X2, Y2, S2).
no_overlap(wcd, X1, Y1, S1, X2, Y2, S2) :-
    -> no_overlap_wcd(X1, Y1, S1, X2, Y2, S2).
no_overlap(cd, X1, Y1, S1, X2, Y2, S2) :-
    -> no_overlap_cd(X1, Y1, S1, X2, Y2, S2).

no_overlap_spec(X1, _Y1, S1, X2, _Y2, S2) :-
    leqc(X1, S1, X2) ? true.
no_overlap_spec(X1, _Y1, S1, X2, _Y2, S2) :-
```
\texttt{leqc(X2, S2, X1)}
\texttt{? true.}
\texttt{no\_overlap\_spec(_X1, Y1, S1, _X2, Y2, S2)} :-
\texttt{leqc(Y1, S1, Y2)}
\texttt{? true.}
\texttt{no\_overlap\_spec(_X1, Y1, _S1, _X2, Y2, S2)} :-
\texttt{leqc(Y2, S2, Y1)}
\texttt{? true.}
\texttt{leqc\_iff(X, C, Y, B)} :-
\texttt{fd(ask, X+C=Y)}
| \texttt{fd(B=0)}.
\texttt{leqc\_iff(X, C, Y, B)} :-
\texttt{fd(ask, X+C=Y)}
| \texttt{fd(B=1)}.
\texttt{leqc\_iff(X, C, Y, B)} :-
\texttt{fd(ask, B=0)}
| \texttt{fd(X+C=Y)}.
\texttt{leqc\_iff(X, C, Y, B)} :-
\texttt{fd(ask, B=1)}
| \texttt{fd(X+C=Y)}.
\texttt{leqc(X, C, Y)} :-
\texttt{fd(Y>=X+C)}.
\texttt{no\_overlap\_card(_X1, Y1, S1, _X2, Y2, S2)} :-
\texttt{domain([B1,B2,B3,B4], 0, 1)}.
\texttt{leqc\_iff(X1, S1, X2, B1)}.
\texttt{leqc\_iff(X2, S2, X1, B2)}.
\texttt{leqc\_iff(Y1, S1, Y2, B3)}.
\texttt{leqc\_iff(Y2, S2, Y1, B4)}.
\texttt{fd(B in 1..4)}.
\texttt{fd(B = B1+B2+B3+B4)}.
\texttt{no\_overlap\_wd\_card(_X1, Y1, S1, _X2, Y2, S2)} :-
\texttt{fd(YA in min(Y1)+S1...)}.
\texttt{fd(YB in 0..max(Y1))}.
\texttt{fd(YC in min(Y2)+S2...)}.
\texttt{fd(YD in 0..max(Y2))}.
\texttt{fd(XA in min(X1)+S1...)}.
\texttt{fd(XB in 0..max(X1))}.
\texttt{fd(XC in min(X2)+S2...)}.
\texttt{fd(XD in 0..max(X2))}.
\texttt{fd(X1 in (dom(YA)/\dom(YD)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(YB)/\dom(YC)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(XC) \setminus (0..max(X2)-S1))}.
\texttt{X2 in (dom(YA)/\dom(YD)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(YB)/\dom(YC)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(XA) \setminus (0..max(X1)-S2))}.
\texttt{fd(Y1 in (dom(XA)/\dom(XD)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(XB)/\dom(XC)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(YC) \setminus (0..max(Y2)-S1))}.
\texttt{Y2 in (dom(XA)/\dom(XD)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(XB)/\dom(XC)) \rightarrow (0...)} \texttt{\lor}
\texttt{(dom(YA) \setminus (0..max(Y1)-S2))}.
no_overlap(X1, _Y1, S1, X2, _Y2, S2) :-
    leq(X1, S1, X2) 
    || true.
no_overlap(X1, _Y1, S1, X2, _Y2, S2) :-
    leq(X2, S2, X1) 
    || true.
no_overlap(X1, _Y1, S1, X2, _Y2, S2) :-
    leq(Y1, S1, Y2) 
    || true.
no_overlap(X1, _Y1, S1, X2, _Y2, S2) :-
    leq(Y2, S2, Y1) 
    || true.

state_capacity(Cs, Sizes, Size, D) :-
    state_capacity(1, Size, Cs, Sizes, D).
state_capacity(Pos, Size, .. .. ..) :-
    Pos=Size
    -> true.
state_capacity(Pos, Size, Cs, Sizes, D) :-
    -> accumulate(Cs, Sizes, Pos, Size, D),
    Pos is Pos+1,
    state_capacity(Pos1, Size, Cs, Sizes, D).

acc(spec, X, S, P, B) :-
    -> acc_spec(X, S, P, B).
acc(card, X, S, P, B) :-
    -> acc_card(X, S, P, B).
acc(wcd, X, S, P, B) :-
    -> acc_wcd(X, S, P, B).
acc(cd, X, S, P, B) :-
    -> acc_cd(X, S, P, B).
acc_spec(X, S, P, B) :-
    fd( B in 1 ),
    PS is P-S+1,
    fd( X in PS..P )
    ? true.
acc_spec(X, S, P, B) :-
    fd( B in 0 ),
    PS is P-S,
    P1 is P+1,
    fd( X in (0..PS)\(P1...) )
    ? true.
acc_card(X, S, P, B) :-
    fd(ask, B=0 )
    | PS is P-S,
    P1 is P+1,
    fd( X in (0..PS)\(P1...) ).
acc_card(X, S, P, B) :-
    fd(ask, B=1 )
    | PS is P-S+1,
    fd( X in PS..P ).
acc_card(X, S, P, B) :-
    PS is S-1-P,
\begin{verbatim}
fd(ask, 0 in max(X)*P..min(X)*PS)
| fd( B = 1 ).
acc_card(X, S, P, B) :-
        PS is S-P.
        P1 is P+1.
        fd(ask, 0 in (O..min(X)-P1) \ (max(X)+P ... ) )
| fd( B = 0 ).

acc_wcd(X, S, P, B) :-
        PS is P-S.
        P$1$ is P$1$+1.
        P1 is P1.
        fd( T1 in (PS1..P) ),
        fd( T2 in ((0..PS)\(P1...)) ),
        fd( B in ((dom(T1)\dom(X)) \ 1 \ (dom(T2)\dom(X)) \ 0) ),
        X in ((dom(B)\0) \ (dom(T2) \ (dom(B)\1) \ (dom(T1)) )).

acc_cd(X, S, P, B) :-
        fd( B in 1 ),
        PS is P-S1.
        P1 is P1.
        fd( X in PS..P )
| true.

acc_cd(X, S, P, B) :-
        fd( B in 0 ),
        PS is P-S.
        P1 is P1.
        fd( X in (0..PS)\(P1... )
| true.

accumulate([], [], [], 0, _).
accumulate([C | Cs], [S | Ss], Pos, Sum, D) :-
        fd( B in 0..1 ),
        acc(D, C, S, Pos, B),
        accumulate(Cs, Ss, Pos, Sum1, D),
        fd( S$\oplus$Sum1$\oplus$Sum ).

B.9 Cars

For a description of the problem see Section 2.3.1. For this problem no particular enumeration is needed, the variables can be assigned in an arbitrary order.

cars(Lab,X) :-
X=[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10].
Y=[011,012,013,014,015,
   021,022,023,024,025,
   031,032,033,034,035,
   041,042,043,044,045,
   051,052,053,054,055,
   061,062,063,064,065,
   071,072,073,074,075,
   081,082,083,084,085].
\end{verbatim}
B.10 Anytime scheduling

For a description of the problem see Section 2.3.2. In this case, the variables are enumerated in reverse ordering of the input order, such that the first solution found is the optimal solution [CF94].

scheduler(M) :- scheduler(M, []).

scheduler([], _) :-
    -> true.
 scheduler([output_sheet(S,L,D)|M], V) :-
    -> constrain_sheet(S, L, O, V1),
        scheduler(M, V1).
 scheduler([schedule_request(O)|M], V) :-
    -> minimize & select(O, A, V, V1),
        write_answer(A),
        scheduler(M, V1).

constrain_sheet(Type, L, O, VO, V) :-
    loop_length(K),
    Z is L - K,
    max(Z, 0, ZK),
    fd(X in (1X*K) .. (2X*L*K+O) ),
    order_constraint(Type, X, K, VO),
    loop_constraint(1, Type, X, K, VO),
    inversion_constraint(Type, X, K, VO),
    V = [{sheet,X,Type}|VO].
B.10 Anytime scheduling

order_constraint(_ _, _, []):-  
  true.
order_constraint(s, X, K, VO):-  
  VO=[(_X1,d)]  
  -> fd( (X+K) < X ).
order_constraint(d, X, K, VO):-  
  VO=[(_X1,a)]  
  -> fd( X1 < (X+K) ).
order_constraint(_, X, _, VO):-  
  VO=[(_X1,)]  
  -> fd( X1 < X ).

loop_constraint(X, 0, K, _):-  
  -> true.
loop_constraint(_ _, _, []):-  
  true.
loop_constraint(I, s, X, K, VO):-  
  VO = [(_s)|V1]  
  -> II is I+1,  
     loop_constraint(II, s, X, K, V1).
loop_constraint(I, d, X, K, VO):-  
  VO = [(_Xi,d)|V1]  
  -> fd( X< X ),  
     II is I+1,  
     loop_constraint(II, d, X, K, V1).
loop_constraint(I, Type, X, K, VO):-  
  VO = [(_Xi,s)|V1]  
  -> fd( X> X ),  
     II is I+1,  
     loop_constraint(II, Type, X, K, V1).

inversion_constraint(_ _, _, []):-  
  true.
inversion_constraint(s, X, K, VO):-  
  VO = [(_X1,d)]  
  -> fd( (X+K+1) \ X ).
inversion_constraint(d, _, _, VO):-  
  VO = [(_X1,a)]  
  -> true.
inversion_constraint(d, _, 1, _):-  
  -> true.
inversion_constraint(d, X, K, VO):-  
  VO = [(_Xi,s)|V1]  
  -> fd( X< X+1 ),  
     K1 is K+1,  
     inversion_constraint(d, X, K1, V1).inversion_constraint(_, X, K, VO):-  
  VO = [_|V1]  
  -> K1 is K+1,  
     inversion_constraint(d, X, K1, V1).
B.11 N-queens with Disjunctive Member

We give the four different versions of the n-queens program which exploits a disjunctive member/2 constraint. The code which common to the standard n-queens is emitted (see Section B.5).

```prolog
queens_spec(N, L) :-
    constructDomains(N, L),
    constrain(L),
    spec_member(1, N, L),
    labeling_ff(L).

spec_member(M, N, _) :-
    M = N
    -> true.
spec_member(M, N, L) :-
    spec_mem(M, L),
    M1 is M+1,
    spec_member(M1, N, L).

spec_mem(M, L) :-
    member(M, L)
    ? true.

queens_card(N, L) :-
    constructDomains(N, L),
    constrain(L),
    card_member(1, N, L),
    labeling_ff(L).

card_member(M, N, _) :-
    M = N
    -> true.
card_member(M, N, L) :-
    atleast(1, L, M),
    M1 is M+1,
    card_member(M1, N, L).

queens_weak(N, L) :-
    constructDomains(N, L),
    constrain(L),
    weak_member(1, N, L),
    labeling_ff(L).

weak_member(M, N, _) :-
    M = N
    -> true.
weak_member(M, N, L) :-
    weak_member(M, L),
    M1 is M+1,
    weak_member(M1, N, L).

wcd_member(M, L) :-
    wcd_mem_bools(L, M, Bs),
    wcd_mem(L, [], Bs, M).
```
B.11 N-queens with Disjunctive Member

\[
\text{wcd_mem_bools}([], \_\_\_, Bs) :- \\
\quad \text{true}.
\]

\[
\text{wcd_mem_bools}([X|L], M, Bs) :- \\
\quad \text{true}.
\]

\[
\text{wcd_mem}([], \_\_\_, \_\_\_).
\]

\[
\text{wcd_mem}([X|L], R, [B]\ [B]\ Bs], M) :- \\
\quad \text{true}.
\]

\[
\text{sum_up}([], S) :- \\
\quad \text{false}.
\]

\[
\text{sum_up}([X|L], S) :- \\
\quad \text{false}.
\]

\[
\text{queens_strong}(N, L) :- \\
\quad \text{true}.
\]

\[
\text{constructive_member}(M, N, \_\_\_) :- \\
\quad \text{true}.
\]

\[
\text{member}(N, [Y|\_\_]\_) :- \\
\quad \text{false}.
\]

\[
\text{member}(N, [\_|L]\_) :- \\
\quad \text{false}.
\]
Computing Science Department, Uppsala University
Uppsala Theses in Computing Science

Swedish Institute of Computer Science
SICS Doctoral Dissertation Series

