Efficient Constraint Solving in Dynamic Languages

vorgelegt von
Diplom-Informatiker
Stephan Frank
aus Berlin

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
Dr.-Ing.

genehmigte Dissertation

Promotionsausschuss:

Prof. Dr.-Ing. Stefan Jähnichen (Vorsitzender)
Prof. Dr. rer. nat. Peter Pepper (Berichter)
Prof. Dr. rer. nat. Ute Schmid (Berichter)

Tag der wissenschaftlichen Aussprache: 05. April 2011

Berlin 2011

D 83

Zusammenfassung

Die Integration von Constraints in eine Programmiersprache ermöglicht die elegante deklarative Beschreibung von Problemen mit unvollständiger Information, z.B. von Such- und Optimierungsproblemen, sowie Aufgaben in den Bereichen Planung, Analyse, Simulation, Diagnose und Testen.

Das Problemlösen durch die Beschreibung mit Hilfe von Constraints, d.h. Formeln, die entsprechende Einschränkungen für Variablen darstellen, hat in den letzten Jahren durch die Entwicklung von mächtigeren und schnelleren Algorithmen an Bedeutung gewonnen. Oft werden zum Experimentieren mit neuartigen Datenrepräsentationen und Algorithmen kommerzielle Implementierungen benutzt, deren Flexibilität in diesen Bereichen jedoch stark eingeschränkt ist.

Ziel dieser Arbeit war daher zum einen die Schaffung eines Constraint-Programmier- und -Experimentier-Systems mit hoher Modularisierung und Flexibilität aber auch einer angemessenen Performanz. Dabei spielt vor allem auch die Realisierung in Hinblick auf die zugrunde liegende dynamische Host-Sprache eine Rolle, die zwar einen komfortablen, experimentierlastigen Entwicklungsansatz erlaubt und begünstigt, was aber zu Lasten der Performanz geht. Die Arbeit behandelt zwei große Aspekte: Zum einen beschreiben wir ausführlich theoretische Grundlagen, Design und Implementierung unseres Solver-Systems CLFD. Zum anderen untersuchen wir die Integration der Constraint-Programmierung als domainspezifische Sprache DSL.

Im *Solver-Framework CLFD* wird die Flexibilität in allen Solverbereichen als Basiselement zugrunde gelegt. Der Benutzer soll jederzeit die Möglichkeit haben aus einem Spektrum verschiedener Basisstrukturen und -funktionen zu wählen und zu experimentieren. Gleichzeitig erlaubt der streng modularisierte Aufbau die einfache Ergänzung durch eigene Implementierungen nahezu aller Solverbereiche. Definierte Protokolle für die Interaktion zwischen den einzelnen Solverelementen sind die Schlüsselelemente für die erzielte Flexibilität und ermöglichen auch die (eingeschränkte) Kontrolle der Einhaltung theoretischer Grundannahmen.

Beispielhaft stellt CLFD bereits eine Anzahl verschiedener Implementierungen z.B. für Integer-basierte Finite Domains zur Verfügung. Außerdem wurden eine Vielzahl von Constraintimplementierungen für diesen Domain-Bereich integriert, bis hin zum automatischen Handling nicht-linearer Ausdrücke und mehrerer globaler Constraints.

Neben den eigentlichen Constraint-Löser-Konzepten diskutiert die Arbeit die Anwendung von Heuristiken zur Lösungsfindung. Auch hier wurde ein möglichst flexibler und durch den Nutzer erweiterbarer Ansatz gewählt. Wir betrachten hierbei zum einen den Schedulerprozess, der die bei der Constraintpropagierung

verwendete Reihenfolge auf Basis verschiedenster Datenpunkte dynamisch steuert. Zum anderen entwickeln wir ein differenziertes und durchdachtes Suchframework zur Realisierung und zum Tuning verschiedenster Suchmethoden und -heuristiken.

Da sehr flexible Methoden und Konzepte zur Modularisierung und Dynamisierung fast zwangsläufig zu einer verminderten Performance führen, stellen wir verschiedene Ansätze zu einer weitgehenden Vermeidung bzw. Verminderung dieser Probleme vor und diskutieren diese im Detail. Dabei gehen wir insbesondere auch auf Aspekte in Design und Implementierung hinsichtlich der dynamischen Hostsprache ein.

Unser Solver-System CLFD ermöglicht die (dynamische) Definition verschiedener Constraintprobleme und erlaubt die einfache Evaluierung verschiedenster Lösungsmethoden und Datenstrukturen.

Ein zweiter Aspekt der Arbeit beschäftigt sich mit der *Sprachintegration* von CLFD. Das Framework in der obigen Form stellt eher einen Constraint-Bibliotheksansatz dar. Constraint-Probleme werden mit Standardsprachmethoden (Instantiierung, Funktionsaufrufe etc.) definiert und evaluiert. Die Benutzung von Constraint-Programmierung wirkt deshalb oft aufgesetzt und sprachfremd. Eine weitere Zielsetzung war daher die Entwicklung einer Domain-spezifischen Sprache (DSL) zur stufenlosen Integration in die zugrunde liegende dynamische Host-Umgebung. Hierbei betrachten wir vor allem zwei Ebenen: Zum einen muss es möglich sein, in möglichst formeller Art und Weise Constraintprobleme zu definieren; zum anderen sollte auch die Erweiterung des Solvers, insbesondere die Ergänzung zusätzlicher Einschränkungsalgorithmen, weitgehend (und soweit möglich) deklarativ erfolgen können.

Abstract

The integration of constraints into a programming language enables the elegant declarative description of problems with incomplete information, e.g. search and optimization problems or task descriptions in the areas of planning, analysis, simulation, diagnosis and testing.

The inference of solutions through problem descriptions with constraints, i.e. formulas which represent respective restrictions for variables, has been on the rise in recent years. This has been driven by the development of powerful and fast algorithms. New data representations and experimentation with newly developed algorithms need a flexible framework to allow the simple integration and adaption. Current systems, especially from commercial vendors, often lack the openness to allow such research comfortably.

The *goal* of this thesis is the development of a constraint programming and experimentation framework which is highly modularized and flexible but which at the same time still provides acceptable performance.

The work is divided in two aspects of work: a first part develops and describes comprehensively the theoretical aspects, design and implementation decisions of our solver framework CLFD. The dynamic nature of the host programming language has been influencing many design aspects and implementation decisions. A second part reflects on the transparent integration of the constraint description into the underlying dynamic host language.

The basic element of all parts of the solver framework CLFD is the flexibility in all areas. A user shall have the possibility to select from a range of base structures and base functions and modularly enhance and experiment with these elements. The modular design allows for an exchange of different implementations in all areas. Defined protocols at the module borders and interactions points are the key element for this flexibility. They also ensure the compliance with the presented module properties (if computationally feasible). Another aspect is the design in relation to the dynamic host language. It allows for a dynamic reorganization and optimization, yet at the same time requires a different design with respect to statically compiled languages to ensure a well performing solver engine.

Exemplarily, CLFD provides a number of different domain and management data structures. Furthermore, several constraint pruning algorithms from simple domain restriction, automatic handling of non-linear constraints to complex global constraints are made available.

Additionally to the actual constraint pruning several concepts for the heuristic extension for solution finding are discussed. We focus on the scheduler process which selects the different constraints in a certain order during the propagation

phase. This selection order greatly influences the overall performance. Furthermore we define a sophisticated search framework which realizes and tunes many aspects of heuristic search in a highly flexible way.

Such a flexible, modularized concept in addition to the dynamic aspects of the host language comes with a performance penalty, though at the same time greatly enhance on-the-fly optimization and experimentation. Therefore we present and detail several approaches to redeem or at least cushion such performance problems.

Our solver framework enables the dynamic definition of different constraint problems and solver modules and thus allows for a comfortable evaluation of different solving methods and data structure aspects.

A second aspect of this thesis is the transparent *language integration* of the constraint system definition and solving properties of CLFD. The solver framework as described above represents an approach quite similar to a programming library. Constraint problems are defined and evaluated by calling a number of respective interface methods (object instantiation and function calls).The employment of constraint programming thus feels quite foreign and superimposed compared to the underlying host paradigm.

A further goal of this thesis is thus to define and evaluate a domain specific language (DSL) approach to enable a transparent integration within the underlying host language environment.

We try to approach this problem on two levels: Firstly, it should be possible to define constraint problems within such a language; furthermore, the DSL should allow for the enhancement and definition of at least some solver aspects in a declarative way.

Contents

1. Introduction

Since their introduction, declarative programming languages have embraced the idea of programs that are as close as possible to their problem domain, thus enabling an abstract description of a problem rather than a detailed step by step algorithmic implementation as is typically necessary with an imperative language approach. Constraint programming is one representative of the declarative paradigm. Generally, a constraint problem is formulated as a mathematical model consisting of relations and functions.

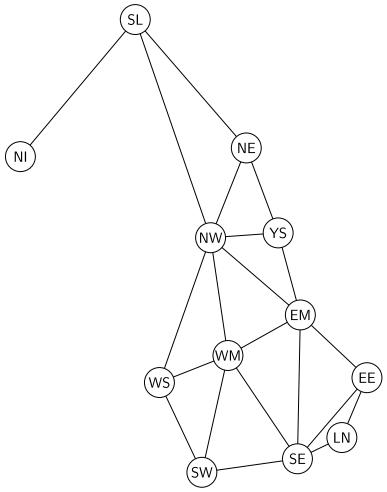
Example 1.1. As a first example of a (finite-domain) constraint problem consider the *map coloring problem*. We want to color the regions of a map such that no two neighboring regions are of the same color. Figure 1.1a depicts a map of the United Kingdom with its countries (Scotland, Whales and Northern Ireland) and the Government Office Regions for England (Northeast, Northwest, Yorkshire, West Midlands, East Midlands, South West, South East, East of England and London). We want to color the map with at most four colors: *red, green, yellow* and *blue*. To formulate the constraint problem we create a variable for each region where every variable is of the domain $\{red, green, blue, yellow\}$. Additionally, we need to express that neighboring regions must be assigned to different colors, i.e. variables representing adjoined regions must have a different domain value assigned to each of them in a feasible solution. Figure 1.1b shows the corresponding *constraint graph* G to our problem. This graph is induced by the variable nodes V_{var}, one for each different map region and the edges E expressing the neighbor-ship relation for any two regions. We can now express the problem as a formula:

$$V_{var} = \{EE, EM, LN, NE, NI, NW, SE, SL, SW, WM, WS, YS\} \wedge \bigwedge_{v \in V_{var}} v \in \{red, green, blue, yellow\} \quad \wedge \quad \bigwedge_{\substack{i,j \in V_{var} \\ (i,j) \in E \\ i <_{lex} j}} i \neq j$$

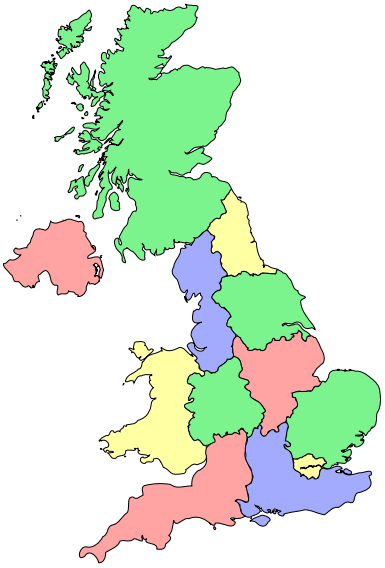
Where $<_{lex}$ is defined as the lexicographical ordering on the variable names.

A solution adhering this constraint conjunction is depicted in Figure 1.1c with the assignment

(a) Map of the UK with countries and Government Office Regions.



(b) The corresponding constraint graph $G = (V_{var}, E)$ expressing the neighboring relation between the different regions as represented by the graph nodes.



(c) A possible coloring of the map.

Figure 1.1.: Example of the 4-color map coloring problem.[a]

[a] The graphics are based on a map from the UK Office for National Statistics (`http://www.statistics.gov.uk/geography/downloads/uk_gor_cty.pdf`).

$$
\begin{aligned}
SL, YS, WM, EE &= green \\
NE, WS, LN &= yellow \\
NI, EM, SW &= red \\
NW, SE &= blue
\end{aligned}
$$

as one of several possible solutions. Also note that a solution with fewer colors would not be possible since West Midlands (WM) is adjoined with an odd number of regions. If there would only be two other colors left for assignment we would always end up with two neighboring regions of the same color and thus a violation of the coloring inequality of adjoining areas.

Problems described by constraints are handled by constraint solvers. These are programs which essentially represent a set of sophisticated algorithms that are able to check for satisfiability of constraint conjunctions and compute implied, i.e. entailed, constraints. Constraint solvers typically operate on a limited application domain because efficient solving algorithms only exist for such confined application areas. We call the programmatic description and construction of constraint problems by means of a programming language *constraint programming*.

Constraint programming is a relatively young member of the family of declarative programming languages though it has seen wide interest and application in recent years. It has its origins in the logic language PROLOG, where constraints were introduced due to limitations when reasoning about mathematical expressions [Dincbas *et al.*, 1988]. A pure logic programming approach necessitates the detailed description of mathematical domains by relations; an approach that quickly becomes costly during evaluation. The evaluation strategy of using search, as is inherent in the logic language paradigm, integrates also very naturally with the constraint solving paradigm, so for some time constraint programming has almost exclusively been tied to logic languages.

Meanwhile, functional-logic languages have embraced and integrated the logic approach, and thus also constraint programming, with CURRY [Hanus *et al.*, 1995] as its most prominent representative.

Constraints have been widely applied to many areas of reasearch, e.g. in the area of *inductive programming* [Schmid and Waltermann, 2004] where functions and transformation are infered from a set of examples, constraints proved to offer alternative approaches [Anthony and Frisch, 1996]. Also, in classical AI areas such as planning and spatial inference [Schmid *et al.*, 2002] were constraints applied succesfully.

1.1. Motivation

To enable constraint programming in the mainstream imperative world a library approach has been pursued, e.g. with the highly successful ILOG Solver [Puget, 1994]. In this model the functions (or methods) for constraint problem definition and search space exploration are encapsulated in a support library. However, the definition of constraint problems in this case is either rather elaborate or falls back to an additional (external) description language that does not smoothly integrate with the host language, e.g. ILOG's OPL Studio or Open Solver [Zoeteweij, 2005].

The approach of the SCREAMER library [Siskind and McAllester, 1993b] for Common Lisp overcomes that impedance mismatch and blends in with the host language. However, the overall approach makes it very difficult to add new constraint domains or even search methods and heuristics to the solver system.

1.1.1. Thesis

In this thesis we will approach two cornerstones of the above systems: First we will present a modular solver design that enables the easy exchange and addition of constraint domains, solving algorithms and solving strategies/heuristics. The system should also support the dynamic definition and re-definition of constraint problems and solver parts as it is a key feature of a mature Common Lisp environment. At the same time it is vital that such dynamism should not come with a high performance penalty. We will show that our design is within an acceptable performance range when compared to statically compiled languages while at the same time exhibiting the development advantages of a dynamic programming environment.

The main thesis of this work is:

> A competitive constraint solver with generic, exchangeable modules can be efficiently implemented in a dynamic language.

We approach this goal by laying out the general properties of a generic constraint solver framework first in a rigorous mathematical way, followed by an assessment from a practical point of view, e.g. necessary (performance) attributes of the used data structures. In a second step we devise a framework model with its accompanying interface protocols. Practical experimentation and benchmarking supports the proposed approach as feasible in practice.

1.1.2. Contributions

The main contributions are:

1. An exploration and explanation of the supporting modules of a generic constraint solver framework. Furthermore an interface *protocol* of generic functions that designate the necessary interaction behavior of the individual components and allow for their easy exchange and thus allow the experimentation with different algorithmic and data structure approaches in the modules. Additionally, we describe several core data structures that support and manage the inter-module cooperation.

2. A mathematical model for the soundness, termination conditions and termination behavior of the essential solver modules, especially the pruning propagators and their application by the responsible scheduler.

3. An implementation of the described framework and realization of its modules (several of them with a number of alternative approaches to illustrate the flexibility). It provides a competitive solver when compared to previous dynamic approaches but also can compete (within limits) with static design approaches.

4. A discussion, motivation and application of a statistical model that is suitable to handle the number of unavoidable noise events in the benchmarking with dynamic languages and their run-time systems.

Basing on the above solver framework we will show a transparent integration of constraint solving as a domain specific language into the underlying host system and language.

We base our implementation on the Common Lisp language and its object-oriented extension the Common Lisp Object System (CLOS). The fundamental design concepts of our architecture do not rely on specific CLOS features but rather make use of common concepts such as reflection, high-order functions, data encapsulation within object instances and run-time code generation. However, the implementation of several of our design abstractions has been greatly simplified by advanced CLOS features such as method combinations [Kiczales and Rivieres, 1991] and the "code is data" philosophy of Lisp, where source code can be analyzed and modified by complex macros before compilation and execution.

1.2. Outline

The thesis is organized as follows: We start in Chapter 2 with an introduction of the necessary definitions that will allow us to talk and reason about constraints and their properties and solutions in general.

Chapter 3 presents the overall architecture of the solver framework. We will first establish the fundamental properties of the individual solver parts before giving and justifying a specific design description in Section 3.1. To encapsulate the different solver parts and to make them easily adaptable, we designed several interface protocols. These protocols are presented in Section 3.2. Furthermore, Section 3.3 shows the actual module implementations that were done to facilitate the flexible foundation of the protocol interfaces.

Propagators are at the heart of the domain reduction phase of a constraint solver. They are discussed in Chapter 4. In Section 4.1 the general consistency iteration and its varying approaches are illustrated. This is followed by an overview of the currently available propagator implementations in CLFD. We implemented several propagators, starting from simple arithmetic expressions, handling of non-linear constraint, to complex global constraints. Eventually, we discuss several enhancements and additions to previous approaches.

Having introduced the necessary constraint propagation parts we will complete the solver framework in Chapter 5 with a presentation of a generic search architecture that is able to express a wide variety of search algorithms with minimal effort. We will discuss the general properties before introducing the different composition blocks: search goals (Section 5.2.1), choice-points (Section 5.2.2) and exploration heuristics (Section 5.2.3).

Given the complete set of solver module abstractions in conjunction with their implementations, Chapter 6 reasons about the necessary conditions for a terminating solver system. It is essential for every module implementation to adhere to the illustrated and proven properties to ensure a terminating solver process.

Finally, Chapter 7 discusses the overall framework performances with respect to our initially set goal of a well performing solver within the bounds of a dynamic programming language. Firstly, Section 7.1 presents the statistical model used in our performance evaluation since system-inherent and thus unavoidable events such as garbage collection and dynamic (re)compilation can have detrimental and misleading effects on the measurements. After detailing the benchmark problems in Section 7.2 we present the performance evaluation in Section 7.3.

To make the features and dynamic archtitecture more accessible to users as well as to ease the development and integration of additional constraint relation, we present a number language integration features which provide to transparently integrate declarative constraint features into the underlying Lisp system.

We conclude our thesis in Chapter 9 and reconsider the achieved results with respect to our initial thesis. Section 9.1 provides an outlook to possible future research topics and enhancements. Finally, Section 9.1.1 discusses several approaches to further reduce search costs by showing how to employ modern CPU design by making use of the multi-core architecture of modern (desktop and server) processors.

2. Background

In this chapter we review and outline the basic concepts and notions of constraints and constraint solving. We start with an overview of dynamic programming languages and their properties, advantages and disadvantages in Section 2.1. Common Lisp as one of the oldest languages with dynamic aspects is the implementation language and backbone of our solving infrastructure. In Section 2.2 we continue by introducing predicate logic as the foundation of constraint (logic) programming. We will introduce necessary definitions and notions of constraint programming in Section 2.3. Finally in Section 2.4, an overview of finite domain constraint solving in general as well as a characterization of related work is given.

2.1. Dynamic Languages

Generally, in computer science a *dynamic programming language* is described as a language that performs and executes behaviors such as type checking at run-time, that other programming languages (actually their compilers) perform during program translation. Furthermore, dynamic languages typically allow the modification of running programs by extending and creating object or class definitions or even enable modifications of the type system [Paulson, 2007; Meijer and Drayton, 2004]. Static languages make such a programming style much more difficult since there is no direct support for such run-time modification.

These abilities allow for shorter turn-around times and compacter programs when developing [Paulson, 2007] but at the cost of performance since the run-time system has to do more work during execution, e.g. type checking and even code generation.

There has been much research in the area of optimization as well as static code identification and static type inference within a dynamic context to enable the widespread use of dynamic language implementations. This started with work on the optimization of Lisp systems [MacLachlan, 1992] but also found recent interest in the research for fast execution of Python [Rigo, 2004] and JavaScript programs [Yermolovich *et al.*, 2009].

However, to profit most from these optimization techniques while at the same time benefiting from the powerful run-time manipulation techniques of the dy-

namic paradigm, programs need to be designed and structured differently [Norvig, 1998]. This subject and its influences on our design will be elaborated during the development of the overall solver framework in Section 3.2.

The key features of dynamic languages are:

Dynamic Typing Most dynamic languages are strongly but at the same time dynamically typed, i.e. it is not possible to apply functions or methods to non-fitting parameters (e.g. summing two strings instead of two numbers) but this test is only done at run-time instead of compile time. This comes at the cost of a run-time overhead but at the same time enables a more dynamic implementation style, where it is possible to run and test program parts during development without already having a complete type-correct program in place. A problematic part of this approach is of course that certain errors will only become apparent during execution where statically, strongly typed languages such as Opal [Pepper, 1991] or Haskell [Jones *et al.*, 1993] enforce these tests at compile time.

Several approaches for additional static analysis at compile time to mitigate this problem and to cushion the run-time overhead have been developed in recent years [MacLachlan, 1994; Siek and Vachharajani, 2008].

Garbage Collection Automatic memory management means that memory is allocated and released by the run-time system. This avoids memory leaks due to unreferenced memory blocks, it overall simplifies and shortens development and brings greater robustness by the reduced risk of bugs due to manual memory handling. Furthermore the lack of pointer manipulation avoids problems due to accidentally/erroneously altered references.

It has been demonstrated that for programs with very complex memory allocation and release behavior, automatic memory management can outperform manual memory handling by the programmer [Jones, 1999].

Experimental Development The typical development style for dynamic languages such as Python or Lisp is to have a run-time system that is incrementally enhanced by the addition and modification of functions, data structures and modules. An interactive prompt (REPL – Read Eval Print Loop) is used for testing and experimentation. This approach allows for a very flexible, exploration directed programming approach which is especially helpful in cases were a best solution is not obvious up-front [Nierstrasz *et al.*, 2005].

Note that this style also favorably blends with our constraint solver framework and additionally provides us with an environment were constraint problems and search heuristics can be defined and solved interactively much like the

interface of computer algebra systems such as Maxima[1] [Bogen *et al.*, 1977], Mathematica [Wolfram, 1991] or the interactive shell of PROLOG systems [Cheadle *et al.*, 2003].

Run-time Reflection Reflection enables the observation of a system's components and behavior during execution. Furthermore, it can be used to alter a system's behavior without the need for a restart. There is typically limited support in static languages for reflection, since it requires elaborate support by the run-time system.

Run-time Modification Reflection is a necessity for run-time modification where objects (or other program data) are observed during execution time and the system alters its behavior accordingly. Several dynamic languages even provide a more powerful way of meta-programming; since all code is also data it is possible to alter even fundamental system attributes such as the object system. For example with the meta-object protocol (MOP) of the Common Lisp object system (CLOS) [Kiczales and Rivieres, 1991] it is possible to alter all aspects of the system's object system such as class storage behavior[2] or the inheritance behavior (even at execution time) in a well defined manner.

To provide these features with reasonable performance the run-time system must be able to translate these changes into efficient machine code. This feature has gained widespread intrest in the form of just-in-time compilation where either a pre-compiled byte-code, as is the case for the JAVA language, or actual source code, e.g. in the case of JAVASCRIPT, is translated to efficient, processor-specific machine code at execution time. However, this process typically does not cover the ability to modify the type system behaviour as is possible within CLOS.

These features individually by themselves are not unique to dynamic languages. However, in their combination they are seen as an advantage when developing solutions to complex problems by the respective language community.

2.2. Predicate Logic and Constraints

Constraints are predicate logic formulas whereas logic languages such as PROLOG (or languages integrating non-deterministic/logic computation) provide an executable

[1] Originally known as MACSYMA.

[2] e.g. transparently adding a persistent store [Kirschke, 1997] or developing a more efficient object instance encoding for a problem at hand

section of predicate logic. Therefore we briefly summarize the foundations of predicate logic and constraints in this section.

A *signature* describes the syntax (i.e. symbols) that can be used to form expressions such as terms or constraints of a language. A corresponding *structure* is responsible for defining the semantic interpretation [Hofstedt, 2001].

Definition 2.1 (signature)**.** A *signature* is a four-tuple $\Sigma = (S, F, R; ar)$ where S is the set of sorts, F the set of function symbols and R the set of predicate symbols. S, F and R are mutually disjoint. The function $ar : F \cup R \to S^*$ denotes the arity function of Σ returning the argument-types. For functions, ar also returns the result-types, where $\forall f \in F : ar(f) \neq \epsilon$ must hold.

The set X^s denotes the set of variables $x_1, \ldots, x_n$ of sort $s \in S$. The set of all variables appropriate to Σ (Σ-*variables*) is simply denoted by X.

Definition 2.2 (Σ-structure)**.** Let $\Sigma = (S, F, R; ar)$ be a signature. A Σ-*structure* $\mathcal{D} = (\{\mathcal{D}^s \mid s \in S\}, \{f^{\mathcal{D}} \mid f \in F\}, \{r^{\mathcal{D}} \mid r \in R\})$ consists of

- an S-sorted family of non-empty carrier sets $\mathcal{D}^s$, where $s \in S$,
- a family of functions $f^{\mathcal{D}}$, where $f \in F$, and
- a family of predicates $r^{\mathcal{D}}$, where $r \in R$.

Thus, for a $f \in F$ with $ar(f) = s_1 \ldots s_n s$ follows that $f^{\mathcal{D}}$ is an n-ary function and $f^{\mathcal{D}} : \mathcal{D}^{s_1} \times \ldots \times \mathcal{D}^{s_n} \to \mathcal{D}^s$ holds. Similarly for a predicate $r \in R$ with $ar(r) = s_1 \ldots s_m$, $r^{\mathcal{D}}$ is an m-ary predicate and $r^{\mathcal{D}} \subseteq \mathcal{D}^{s_1} \times \ldots \times \mathcal{D}^{s_m}$ holds.

Terms allow the description of operations on a syntactical level. Their inductive nature allows the use of the familiar principle of proof by structural induction. A term is either a variable $x \in X$ or it consists of a function symbol $f \in F$ and further terms as its arguments.

Definition 2.3 (set of terms)**.** The *set of terms* $\mathcal{T}(F, X)$ *over* Σ *and* X is defined as follows: $\mathcal{T}(F, X) = \bigcup_{s \in S} \mathcal{T}(F, X)^s$, where for every sort $s \in S$ the set $\mathcal{T}(F, X)^s$ of terms of sort s is the least set containing

1. every $x \in X^s$ (of sort s) and every 0-ary function symbol $f \in F$ with $ar(f) = s$, and
2. every $f(t_1, \ldots, t_n)$, $n \geq 1$, where $f \in F$, $ar(f) = s_1 \ldots s_n s$, and every $t_i, i \in \{1, \ldots, n\}$, is a term in $\mathcal{T}(F, X)^{s_i}$.

Terms without elements of X are called *ground terms*.

Informally, constraint problems consist of variables, domains and constraints. To describe constraint problems we first need to consider *variables* and their associated *domains*. The variables are used as placeholders for values (or set of values) to describe the overall constraint problem.

Domains are used to describe a restricted set of values a variable is a placeholder for. The used definitions conform to a great part to or are based on the standard definitions found in the constraint programming literature, specifically [Apt, 2003; Marriott and Stuckey, 1998].

Definition 2.4 (domain)**.** A (finite) *domain* is an ordered set of (a finite number of) values. Consider a set of variables $X := x_1, \ldots, x_n$ where each variable x_i (where $1 \leq i \leq n$) is bound to a set of values. The domain D maps each variable x_i to the values that x_i is allowed to take, written $D(x_i)$. We will abbreviate $D(x_i)$ by D_i if it is clear from the context which variable from an enumerated set is meant.

We write $\mathcal{D}$ for the domain of all variables, i.e. $\mathcal{D} := D_1 \times \cdots \times D_n$. The *empty* or *false* domain is denoted by $\bot$.

Finite domains are not restricted to sets of numbers, but can also contain more complex value descriptions like graphs [Dooms *et al.*, 2005] or maps. Other areas of constraint solving do not restrict their domains to finite sets but operate on intervals of real numbers [Hickey *et al.*, 2001] or infinite sets [Bodirsky, 2004]. Such infinite domains cannot generally be handled within our solver framework as it is tailored to finite domain constraints and infinite constraints require a completely different algorithmic approach. However, we will later see that it is possible to handle and solve interval constraints within limits since they are quite similar to finite domain constraints even though their domains are not countable.

Primitive Constraints Constraint solvers provide a number of *primitive constraints* (also called *basic constraints*) that work on the variable domains. More complex constraints can be constructed by combining these primitive constraint relations. Typically, such primitive relations are binary but are not limited to and require domains and numbers as their arguments. Examples are $=$ or $\leq$ as relation symbols and $D(x_i) \leq 3$ as an actual primitive constraint expression.

Definition 2.5 (primitive constraint)**.** A *primitive constraint* (or *basic constraint*) is a string $r(t_1, \ldots, t_n)$ with $ar(r) = s_1 \ldots s_n$, $r \in R$ and $t_i \in \mathcal{T}(F, \Sigma)^{s_i}$.

Complex *Constraints* are built by combining primitive constraints with a conjunctive connective denoted by $\wedge$.

Definition 2.6 (constraint)**.** A *constraint* is of the form $c_1 \wedge \cdots \wedge c_n$ where $n > 0$ and the c_i are primitive constraints.

For the finite domain constraints we consider it intuitive to describe constraints from a domain point of view: Consider a finite sequence of variables $X := x_1, \ldots, x_n$ $(n > 0)$ and their respective associated domains $D_1, \ldots, D_n$. A *constraint* C on X is a subset of $D_1 \times \cdots \times D_n$. We denote with $var(C) = X$ the set of variables a constraint C is defined on.

The latter perspective is also valid for basic constraints, however, we will see later that it is advantageous to make the distinction between primitive and normal constraints.

Constraints are interpreted according to the semantics of predicate logic [Ehrig *et al.*, 2001]. We therefore can translate its validity to the Boolean values `true` and `false`.

Definition 2.7 (conjunction)**.** A *conjunction* $C_1 \wedge C_2$ of two constraints C_1 and C_2 is defined as the conjunctive composition $c_1 \wedge \cdots \wedge c_n \wedge c'_1 \wedge \cdots \wedge c'_m$ of the respective primitive constraints where C_1 is $c_1 \wedge \cdots \wedge c_n$ and C_2 is $c'_1 \wedge \cdots \wedge c'_m$.

Having established all the necessary notions we are now finally able to express a constraint (satisfaction) problem which is an interconnection of all the previously introduced elements.

Definition 2.8 (constraint satisfaction problem)**.** A *constraint satisfaction problem* (CSP) consists of a set of variables $X := x_1, \ldots, x_n$ and their respective domains $\mathcal{D} := D_1, \ldots, D_n$ and a constraint C. The CSP represents the constraint $C \wedge x_1 \in D_1(x_1) \wedge \cdots \wedge x_n \in D(x_n)$.

Often the constraint C is also represented as a set of constraints $\mathcal{C} := C_1, \ldots, C_m$ where the constraints C_i $(1 \leq i \leq m)$ are implicitly connected by a conjunction.

We will describe a CSP P as the three-tuple $(C; X; \mathcal{D})$.

Let x_i be a variable then we call $x_i \in D_i$ a *domain expression*. A *solution* of a constraint satisfaction problem is a sequence of values for all its variables such that every constraint in C is satisfied.

Definition 2.9 (solution)**.** Consider a CSP $(C; X; \mathcal{D})$ with $X := x_1, \ldots, x_n$ and $\mathcal{D} := D_1, \ldots, D_n$. An n-tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ *satisfies* a constraint c on the variables $x_{i_1}, \ldots, x_{i_n}$ if $(d_{i_1}, \ldots, d_{i_n}) \in c$. The n-tuple $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$ is a *solution* of the CSP if it satisfies every constraint $c_k \in C$.

Definition 2.10 (valuation domain)**.** A domain $D := D_1, \ldots, D_n$ where every D_i

is a singleton domain is called a *valuation domain*.

Constraint satisfaction problems with a solution are said to be *consistent*, otherwise, i.e. no solution exists, they are said to be *inconsistent*.

Definition 2.11 (equivalence of CSPs)**.** Let $P_0, \ldots, P_n$ $(n \geq 1)$ be constraint satisfaction problems and let X be the set of their common variables (i.e. X is a subset of the variables of each $P_0, \ldots, P_n$). The *union of* $P_1, \ldots, P_n$ *is equivalent w.r.t.* X *to* P_0 if:

1. For every solution d to a P_i (with $i \geq 1$) there exists a solution to P_0 that matches with d on the variables in X.

2. For every solution e to P_0 there exists a solution to some P_i (where $i \geq 1$) that matches with e on the variables in X.

That means that constraints are equivalent, if they have the same set of solutions. The more general approach of Definition 2.11 additionally allows us to describe equivalence for constraints that are split up into sub-constraints which may not contain all variables of the original constraint.

2.3. Consistency

Generally, a *constraint solver* is a set of tests and operations on constraints. Such a solver is able to determine whether a constraint problem is satisfiable or not. Furthermore it is often possible to infer concrete solutions. Solvers for constraint satisfaction problems employ several consistency techniques to test solver operations. Such consistency techniques are used to reduce the domain space of a variable (or a number of variables). We distinguish a number of particular consistency definitions derived from standard literature such as [Apt, 2003].

Definition 2.12 (node consistency)**.** A constraint C is *node consistent* with domain $\mathcal{D}$ if the number of variables of C is not equal to one or, if $vars(C) = \{x\}$, then for every $d \in D(x)$, $\{x \mapsto d\}$ is a solution of C. $vars(C)$ denotes the set of variables contained in constraint C.

A constraint conjunction $C_1 \wedge \ldots \wedge C_n$ is *node consistent* if every C_i, $i \in \{1, \ldots, n\}$, is *node consistent*; that is a CSP is *node consistent* if every unary constraint of C is *node consistent*.

Definition 2.13 (arc consistency)**.** A constraint C is *arc consistent* with domain $\mathcal{D}$ if its number of variables is not equal to two or, if $vars(C) = \{x, y\}$, then for

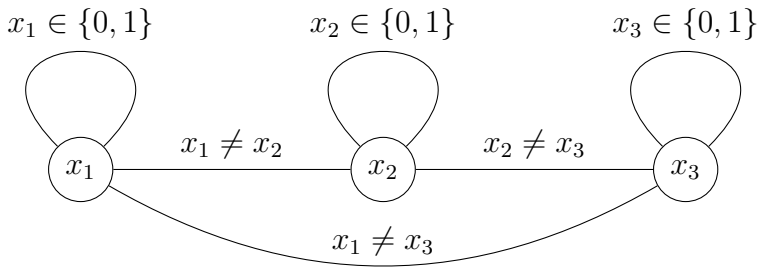


Figure 2.1.: Constraint graph of the arc consistent CSP in Example 2.1.

each $d_x \in D(x)$ there is some $d_y \in D(y)$ such that $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of C and vice versa for every d_y there is some d_x with the same property.

A constraint conjunction $C_1 \wedge \ldots \wedge C_n$ is *arc consistent* if every C_i, $i \in \{1, \ldots, n\}$, is *arc consistent*.

Example 2.1 (An arc consistent constraint)**.** Consider the following CSP: $x_1, x_2, x_3 \in \{0,1\}$, $x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3$. Figure 2.1 depicts a graphical representation of the constraint problem. The constraints are represented by the edges; unary constraints as loops, binary constraints as node-connecting edges.

This CSP is arc-consistent by our definition because for every constraint c of the CSP on two variables x_a, x_b arc-consistency holds: for every domain value of variable x_a there is a domain value of x_b such that c holds and vice versa.

Hyper-arc-consistency is the obvious extension to n-ary constraints with a higher number of participating constraints than two.

Definition 2.14 (hyper-arc consistency)**.** Given a CSP $C = C' \wedge x_1 \in D(x_1) \wedge \ldots \wedge x_n \in D(x_n)$ with $C' = c_1 \wedge \ldots \wedge c_k$, $X := x_1, \ldots, x_n$.

A constraint c with $var(c) = \{y_1, \ldots, y_m\} \subseteq X$ of a conjunction C' is *hyper-arc consistent*, iff for every y_i and for every domain element $d_i \in D(y_i)$ there exists for all y_j, $j \neq i$ an element $d_j \in D(y_j)$ such that c is satisfied, i.e. for every d_i there must be a possible valuation domain D for all other variables such that c is satisfied.

The CSP C is *hyper-arc consistent* if all its constraints $c_1, \ldots, c_k$ of C' are locally consistent.

All of the above notions are called *local consistencies*. For a CSP that is locally consistent *all* domain values must fulfill at least one of the consistency notions.

Definition 2.15 (local consistency)**.** The CSP C is *locally consistent* if all its

constraints $c_1, \ldots, c_k$ are node, arc or hyper-arc consistent respectively.

The local consistency notions cannot guarantee a globally consistent CSP, since their influence is only limited to a subset of the given variables. Consider again Example 2.1 which is arc-consistent. However, since all of the three variable domains contain the same two numbers $\{0, 1\}$ clearly there is no consistent valuation domain. Therefore, a solver that only applies local consistency techniques will in general not be able to find all inconsistencies or all solutions. We call such a solver *incomplete*.

2.4. Finite Domain Constraint Solvers

Enumerating all possible valuation domains and then testing for local consistency is a possible though not feasible approach to derive a consistent solution. This technique is called *generate and test* but has the obvious problem that even few variables with relatively small domains will result in a prohibitively large number of cases to test due to combinatorial explosion. Clearly, we want to reduce domains (and thus search space) as much as possible before enumerating any valuation domains for further testing.

Since local consistency is not complete, a (finite domain) constraint solver applies a *propagate-and-branch* cycle. During the propagation phase the solver applies a pruning function which may remove a number of domain values (and only such values) that are not part of any solution. Such a pruning function is called a *propagator* or *domain reduction function* (DRF) [Zoeteweij, 2005]. In Chapter 4 we give a formal definition of propagators and their properties.

The propagation phase is interleaved with a branching or splitting phase. The result is the *search tree* of the CSP.

Definition 2.16 (search tree [Apt, 2003])**.** The *search tree* of a CSP P with a set of variables X is a (finite) tree such that

- every node is itself a CSP,
- the root node is P,
- nodes at even levels (i.e. 0, 2, ...) have each only one direct descendant (which is derived by constraint propagation)
- on odd levels: if the nodes (and their CSPs) $P_1, \ldots, P_k$ are all direct descendants of a node P_0 then $\bigcup_{i=1}^{k} P_i$ is equivalent to P_0 with respect to X.

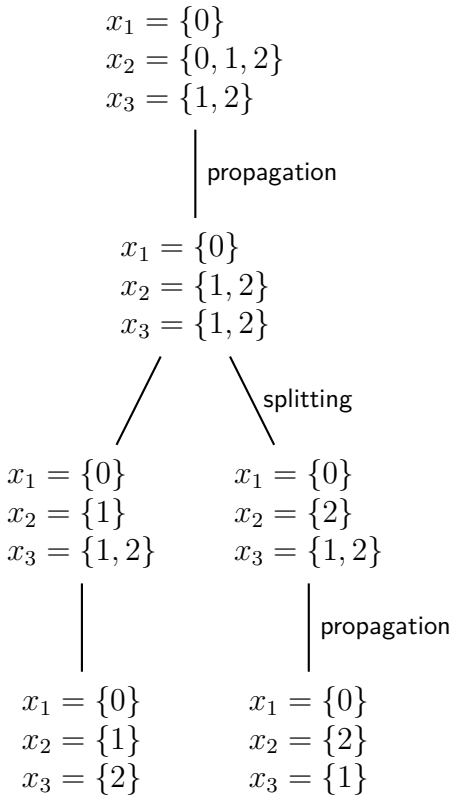


Figure 2.2.: Search tree for the CSP of Example 2.2.

- a node is a leaf node of the search tree either if X has an associated valuation domain or if at least one variable is assigned to the empty domain $\bot$.

Example 2.2 (search tree)**.** Consider the following CSP: $x_1 \in \{0\}$, $x_2 \in \{0, 1, 2\}$, $x_3 \in \{1, 2\}$, $x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3$. In the first round of constraint propagation the 0 is removed due to $x_1 \neq x_2$. The derived CSP is now locally consistent. To eventually get a valuation domain, splitting is now performed in the next step (by arbitrarily selecting and splitting on x_2) resulting in two alternative singleton domains for x_2. Another propagation step yields a solution for each branch. The resulting tree is depicted in Figure 2.2.

The nodes at the odd levels are derived by splitting up a CSP P_0 at an even level into a number of sub-problems by a splitting (or branching) function (remember Definition 2.11 which ensures that no solutions is lost in this way).

Thus at alternating levels we apply constraint propagation or the branching function to generate the search tree. The internal nodes are generated incrementally.

Thus the branching function determines which sub-problem of the search-frontier is explored next.

The leaf nodes of the search tree are either (failed) inconsistent CSP or consistent CSP with a valuation domain, i.e. denote solutions.

If a node results in an inconsistent sub-problem it is necessary to re-start exploring other sub-problems from a higher level to actually find a solution. This is called *backtracking*. The order in which new sub-problems/sub-nodes are selected for further evaluation is subject to a *selection function*:

- Managing and selecting the nodes generated by the branching function in a stack results in a *depth-first search* order.
- Managing and selecting the nodes in a queue results in a *breadth-first* search order.

Apart from these two general handling methods there also exists hybrid and randomized exploration techniques. The search framework, its properties and instantiations are discussed in detail in Chapter 5.

2.5. Related Work

The declarative testing and solution finding in complex systems of interconnected variables and items has been one of the goals in computing from almost the very beginning. Constraint solving has first been established as part of artificial intelligence research but has by now found its way into many application areas such as scheduling, route finding and business rule implementation. In [Rossi *et al.*, 2006] several examples of real world applications are presented.

One of the earliest applications making use of constraint solving has been Sketchpad [Sutherland, 1963]. It was a vector drawing application that used constraints to manipulate and influence the placement of individual drawing elements.

Many different constraint solver systems have been developed within the last decades. At first, almost exclusively as an integral part of Prolog systems due to the close relation of constraint and logic programming yielding constraint-logic programming.

The best known and most influential such systems are Chip [Dincbas *et al.*, 1988], Sicstus Prolog [Carlsson, 2001] and Eclipse Prolog [Wallace *et al.*, 1997; Cheadle *et al.*, 2003]. All implementations are sophisticated logic systems of high performance. The constraint libraries for Prolog systems are often implemented in C for performance reasons and to make direct use of run-time system features. This approach leads to very efficient solvers. However, in general the solver component is

closed and there is only a limited possibility to influence or exchange implementation details such as the propagation schedule. On the other hand, initial designs and solvers for complex domain types such as sets (in the mathematical sense) are implemented directly in PROLOG since it simplifies development by making use of the declarative language features.

In a different user domain, where the introduction of logic languages was not feasible (either due to cultural or impedance mismatch concerns), a number of constraint programming libraries have emerged. The most notable such library is ILOG[3] SOLVER library for C++ [Puget, 1994], a successor to the Lisp-based PECOS system [Puget, 1992]. SOLVER is an extensive library that has widely been used commercially. The accompanying OPL STUDIO provides a declarative domain specific modeling language as a deployment simplification [Van Hentenryck, 1999].

With the spreading use of the JAVA programming language notable constraint libraries are KOALOG[4] [Koa, 2004] and FIRSTCS [Hoche *et al.*, 2003].

All of the above systems have in common that they have been developed over many years in a commercial setting. There have been publications about individual aspects of the solvers, especially specific pruning algorithms, however, literature about the overall design aspects of all necessary solver parts or decisions about individual data structures has long been scarce.

In [Henz *et al.*, 1999] an initial design of a generic (finite domain) constraint solver design and the necessary characteristics, especially from an implementation point of view, has been described. The main focus of FIGARO is on programmable state restoration policies by using the abstraction concept of a room. The configuration possibility is on one very low-level aspect of the solving process. To our best knowledge the project has not been progressed since this initial work.

The presentation in [Schulte and Carlsson, 2006] is to our knowledge the first comprehensive analysis and design description of finite domain constraint solving systems. This work eventually led to the development of GECODE [Schulte and Stuckey, 2008; Tack, 2009], a comprehensive solver library for finite domain constraint solving implemented in C++. It is designed for very high efficiency both in execution speed and memory usage by employing a wide range of low level optimization and template programming techniques. GECODE is currently regarded as the best, widely and freely available solver library that outperforms SOLVER for many test cases.

FACILE [Barnier and Brisset, 2001] is a constraint programming library in OCAML. Making extensive use of the functional programming paradigm it constructs CSPs by functional composition. It is to our knowledge the only solver

[3] now a subsidiary of IBM

[4] as of April 2009 bankrupt and closed

implemented in a functional language. Due to the functional approach adding new constraints is (almost) only a matter of adding a new pruning function. The search method is limited to DFS and not highly customizable. However, FaCiLe has demonstrated very good performance in a demanding setting.

The OpenSolver system [Zoeteweij, 2005] was designed to enable the building of a constraint solver systems that is able to attack a wide range of problems made up by constraints from several different domains. This includes potentially unbounded domains such as $\mathbb{Z}$. It couples a number of black-box solvers integrated through an external configuration language. The implemented and cooperating solvers support only a basic though heavily optimized set of constraints but do not include sophisticated global constraint pruning. Similar work has been proposed and theoretically analyzed [Hofstedt, 2001; Hofstedt and Pepper, 2006] and implemented [Frank *et al.*, 2003b; Frank *et al.*, 2004] in our Meta-S system, though the current implementation of Meta-S does not include a comparable wide range of different domains and optimized solvers but on the other hand offers extended techniques such as language evaluation as constraint solving.

Choco [Laburthe, 2000] is the only commercially developed solver were a detailed architectural description is published. It has since been ported to the Java language and released in source form. Choco offers a wide range of finite domain constraints and an additional unique failure explanation module (Palm) that helps with the analysis of failing models. While most finite domain solvers are propagator based in the sense that a variable change will result in the scheduling for re-execution of all constraints/propagators the variable participates in, Choco is variable based. This means that upon variable change all involved propagators are executed immediately. This requires a different scheduling approach and alternate data structures.

A different strategy for solving CSPs has been proven viable in the Sugar solver [Tamura *et al.*, 2009]. It re-encodes the constraint satisfaction problem into an equivalent Boolean satisfiability (SAT) problem and uses a SAT-solver to derive a solution.

For dynamic languages the Screamer project was an attempt at implementing a non-deterministic sub-language on-top of Common Lisp and an added constraint solving component using this non-deterministic layer [Siskind and McAllester, 1993a; Siskind and McAllester, 1993b]. Screamer makes heavy use of the powerful rewriting capabilities of the Lisp macro system. Non-deterministic code is transformed into continuation-passing-style (CPS) form [Sussman, 1975] which is then used for a Prolog-like search exploration. Initially the backtracking/restoration capabilities have only been implemented for numeric values, but was later extended for other base types such as lists, structures and arrays [White and Sleeman, 1998; White, 2000]. The complex macros especially in interaction with the constraint solver functions and methods make new extensions such as new constraints and

domain-types very difficult. Furthermore, the CPS-conversion implicitly restricts to a depth-first search approach. The assumption that variables are all of the common type `number` and not a more specific type such as `float` or `integer` make a heuristic specialization for specific variable type sets very involved since all numbers types must always be handled.

Our motivating goal for the CLFD framework presented in this thesis, was to show that it is possible to design an open, equally or better performing solver that is also very modular and thus easily extensible at the same time in the context of the dynamic language Lisp. Common Lisp provide us with the means for a highly modular and dynamic constraint solving framework. Since performance is critical for a constraint solver though not as simple to achieve in the context of dynamic languages (cf. Section 2.1) we developed several approaches to leverage this dynamic adaption properties for performance gain within the solver context.

3. The CLFD Solver Framework

This chapter describes the overall design, concepts and functionality of our system. We will first take a high-level look at the system architecture in Section 3.1. Afterwards the individual modules are illustrated in more detail in Section 3.2. At first, the respective interface protocols are presented. Later on in Section 3.3 we will reflect on their specific implementation details.

During our work on the Meta-S system [Frank *et al.*, 2003a; Frank *et al.*, 2005] we noticed a serious lack of available (finite domain) solvers that are easy to interface, for inclusion as a plugable black-box solver. Even more importantly there were only a very limited number of publications about overall solver design. As a result the idea of designing and documenting a generic finite domain solver lead to the initial development of CLFD.

Common Lisp has had constraint solving with the Screamer library [Siskind and McAllester, 1993b] since the early nineties. Screamer allows a quite transparent integration of constraint solving into Common Lisp programs. Unfortunately, the design of Screamer does not welcome the integration of new complex propagators, different heuristics and search strategies, or even alternative domain representations. A reduction method is required to handle all `number` sub-types, thus additionally complicating the implementation of powerful but complex pruning functions, especially for finite domains. Furthermore, its internal workings are rather tied to the binding and bounds propagation model Screamer bases on. The heavy reliance on intricate code walking to achieve continuation passing style (CPS) execution leads to additional complications, when modifying the code base.

Meta-S required a sophisticated, iteratively working (finite-domain) solver. Due to the lack of such solvers that were available and also relatively easy to interface we fell back to the use of rather simple finite-domain solvers which substantially lacked in domain pruning capabilities. CLFD aims at a highly modular design of the overall solver framework that makes it easily extensible through a set of well-defined protocols to every sub-module. The general structure and specified interfaces allow for the simple integration of CLFD into higher-level host systems.

The core part of a (finite domain) constraint solving framework is a constraint propagation engine. It schedules and executes so called propagators to eventually determine a solution. *Propagators* are algorithms that test and remove values from a variable domain, which are not part of any possible solution. Thus a

propagator implements a specific constraint. The specific propagator instances model a complete CSP.

In Chapter 2 we have established the necessary definitions and theoretical foundations to describe constraint systems. Some required elements for an actual solver implementation are already apparent from this theoretical base. To identify and clarify the overall solving scheme and its elements we will consider the following small example.

Example 3.1 (A simple arithmetic relation)**.** Consider a small constraint satisfaction problem with three variables x_1, x_2 and x_3. The initial basic constraints shall determine the following domain values $x_1 \in \{2, 3\}$, $x_2 \in \{4, 5, 6\}$ and $x_3 \in \{5, 6, 7, 8\}$. Furthermore, the constraints $x_1 + 3 < x_2 \wedge x_2 \leq x_3 \wedge x_3 \neq 6$ are required to hold. Figure 3.1 displays the induced CSP-Graph of the problem. It underlines the connection and interdependency of the three constraints through the participating variables.

A propagator scheduler may choose to apply the simple constraint $x_3 \neq 6$ first. This determines the new domain $x_3 \in \{5, 7, 8\}$. The propagation algorithm for $x_1 + 3 < x_2$ can derive that $x_1 \neq 3$ since $6 < x_2$ is not true for any domain value of x_2. This also leads to the inference of $x_2 \neq 4$ as $x_1 < 1$, as well as $x_2 \neq 5$ since $x_1 < 2$ would be required. The resulting domains values are thus $x_1 \in \{2\}$ (or $x_1 = 2$) and $x_2 \in \{6\}$. Finally, the propagator $x_2 \leq x_3$ can deduce $x_3 \neq 5$ since $6 \not\leq 5$. This results in the new domain $x_3 \in \{7, 8\}$. At this point it is not possible for any of the participating constraints to infer any further domain reductions. However, the problem is not solved yet, since x_3 is not bound to a singleton domain. Note that we have chosen a very efficient order for the constraint application. A solver may chose a different ordering which can result in a longer (or shorter) propagation cycle. The heuristic approaches and their technical support foundations will play an essential role in our framework design.

It is now necessary to apply search for actual solution discovery. Search for solutions is started by splitting the domain of some not instantiated variable into two (or more) disjoint subsets and applying constraint propagation on the resulting sub-problems.

In our example we split the domain of x_3 into the two sub-problems $x_3 = 7 \vee x_3 = 8$. For both valuations the propagators test whether $5 < 6 \wedge 6 \leq 7 \wedge 7 \neq 6$ and $5 < 6 \wedge 6 \leq 8 \wedge 8 \neq 6$ holds respectively. The constraints are true for both choices and the solver can determine the two solutions $x_1 = 2, x_2 = 6, x_3 = 7$ and $x_1 = 2, x_2 = 6, x_3 = 8$.

This example illustrates the necessary steps for solution finding: interleaving propagation and search until every variable is assigned to a valuation domain.

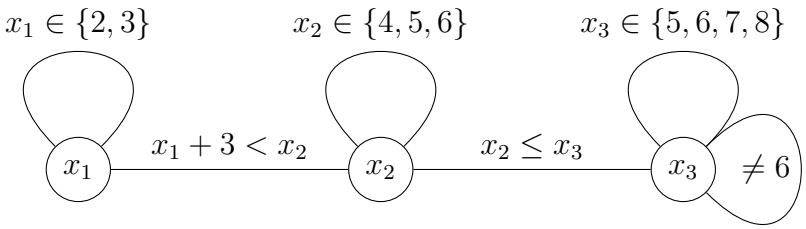


Figure 3.1.: Constraint graph for the CSP in Example 3.1.

From the definitions in Chapter 2 and the CSP example we can infer the following abstractions or elements for CSP representation and constraint propagation:

(Finite) Domains The domain abstraction is responsible for the description of the set of allowed values.

Variables Variables reference a domain thereby representing a *mapping* to a set of (allowed) values, i.e. a variable acts as a placeholder for the domain values until the final valuation domain is derived.

Propagators A propagator represents a constraint and implements the actual domain reduction and consistency testing algorithm. This requires the existence of one propagator for each constraint type in the CSP that relates at least two variables. Unary constraints are expressed by direct actions on the domains as so-called *primitive constraints*.

Stores A store represents a collection of constraints (propagator instances), variables and domains at a certain point of time and thus is a managing data structure to describe a complete CSP.

After the propagation stage a number of sub-CSPs is created to generate a search tree through the interleaving propagation-search stages. The defining item of the search stage is thus a method for domain splitting.

Search Tree Explorer This evaluator takes a number of non-fixed variables and generates a number of sub-CSPs by splitting the domain of one variable into n sub-domains. In the simplest case this is achieved by taking the m domain elements of a single variable and assigning each element to a sub-CSP, which thus implies that $m = n$. We will later see that other splitting methods are possible as well. The explorer must be able to classify derived nodes (solution nodes, intermediate nodes etc.) and progress the search accordingly.

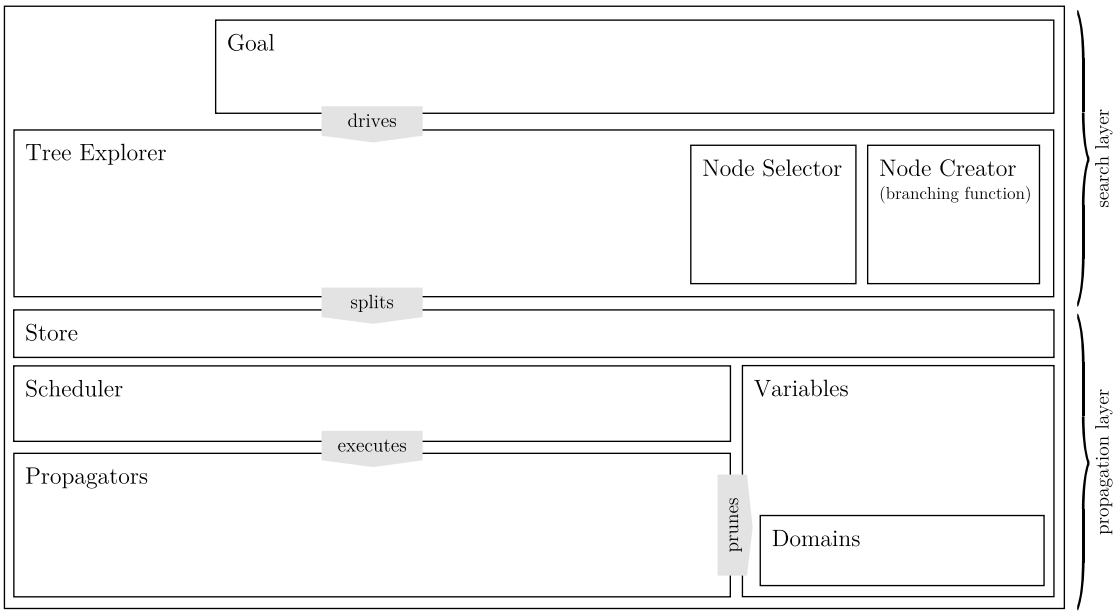


Figure 3.2.: The layered structure of the solver framework.

While this is the general layout architecture, we will later see that a more fine-grained abstraction is necessary to obtain a well performing solver system.

In Figure 3.2 the individual layers and components of the solver framework are displayed. Furthermore, an illustrating outline of their interaction is given. Again, the previously discussed two-layer splitting becomes obvious: a propagation layer is responsible for constraint propagation while a second, higher layer ensures completeness by driving the search coordination.

The tree explorer uses heuristic techniques for node selection and an additional node creation method to generate and explore the search tree. It is guide by a goal observer that dictates what constitutes a solution and what happens with explored leaf nodes.

In correspondence to [Hofstedt, 2001] the complete structure that encodes a CSP is denoted a *configuration*.

Algorithm 3.1 gives a more formal description of the solver operation. It describes the most general and simplest operation of the propagate-and-search process. The actual solver employs far more sophisticated techniques to cut down on computation time. Again, the two phases of propagation and search are clearly visible by the inner loop for propagator execution and an outer cycle for the creation and additional propagation for each resulting individual store. The initial set of stores $\mathcal{S}$ is the singleton set that only includes the initial store s_1. This initial store defines the root of the search tree after the initial propagation phase in the tree exploration

Algorithm 3.1: The basic algorithm of the interleaved propagate-search solving process.

Data: $\mathcal{P}$ is the set of propagators.
Data: $\mathcal{S}$ is the set of stores initialized as singleton set with starting store $\{s_1\}$.
Data: $\mathcal{D}$ is the domain of all variables (cf. Definition 2.4).

```
1  while S ≠ ∅ do
2      foreach s ∈ S do
3          repeat
4              foreach p ∈ P do
5                  propagate p thus reducing D
6              end
7          until no changes to D occurred during the last cycle ;
8          if D is a valuation domain ∨ D = ⊥ then
9              save s
10         else
11             split s and add the resulting s_i to S − {s}
12         end
13     end
14 end
```

for an actual solution.

3.1. The General Module Architecture

The general components of a constraint solver have already been explored in the previous section.

CLFD's underlying design was mostly influenced by the Figaro system [Henz *et al.*, 1999] which in turn was influenced by the Mozart-Oz [Müller and Müller, 1997] programming language and system. The main architectural aspects of CLFD were initially presented in [Frank and Hofstedt, 2005].

Our constraint solver is based on the *propagate*-and-*branch* scheme as described above. The overall architecture can therefore be divided into two parts: a *constraint propagation* component and a *search* module. This separation has already been illustrated in Figure 3.2. The following customizable and exchangeable modules are part of the CLFD system:

Propagation Module: The constraint propagation part consists of the following modules.

Variables and **Domains** encoding the domain values.

Propagators which interconnect the variables by algorithmically testing and enforcing constraints on them.

Schedulers A scheduler is responsible to select the propagator that is executed in the next propagation step. An intelligent schedule can effectively reduce propagation complexity.

Since propagation alone is incomplete a backtracking search is in general necessary to compute solutions. It is desirable to employ search strategies and heuristics which are specific to a particular problem class.

Search Module: The incremental creation and exploration of the search tree is a second area that greatly influences the performance of the solving process. To enable the use of different search approaches and heuristics which are tied to and beneficial for a specific problem class, we provide the following abstractions.

Search Node Container A node represents a choice-point, i.e. a node in the search tree. It holds the information necessary to navigate the computation of successor nodes and to restore the store for this specific node (e.g. by holding a copy, or by handling recomputation information).

Branching A branching encapsulates the information that is used to create the next-level nodes of the search tree from the current one.

Search Strategy The tree visiting method that controls the order of the node visits.

Goal Descriptor The highest level abstraction of the search algorithm. It determines especially if or how a search resumes or stops after a solution or fail node was encountered.

The above components together describe (and eventually solve) a CSP. It is called a *configuration* in our system. Figure 3.3 graphically illustrates the general structure of the overall propagation architecture and the individual dependencies. The components of the search layer a depicted in Figure 3.4. In the following section we will discuss the individual components and argue about their necessary properties and the resulting design.

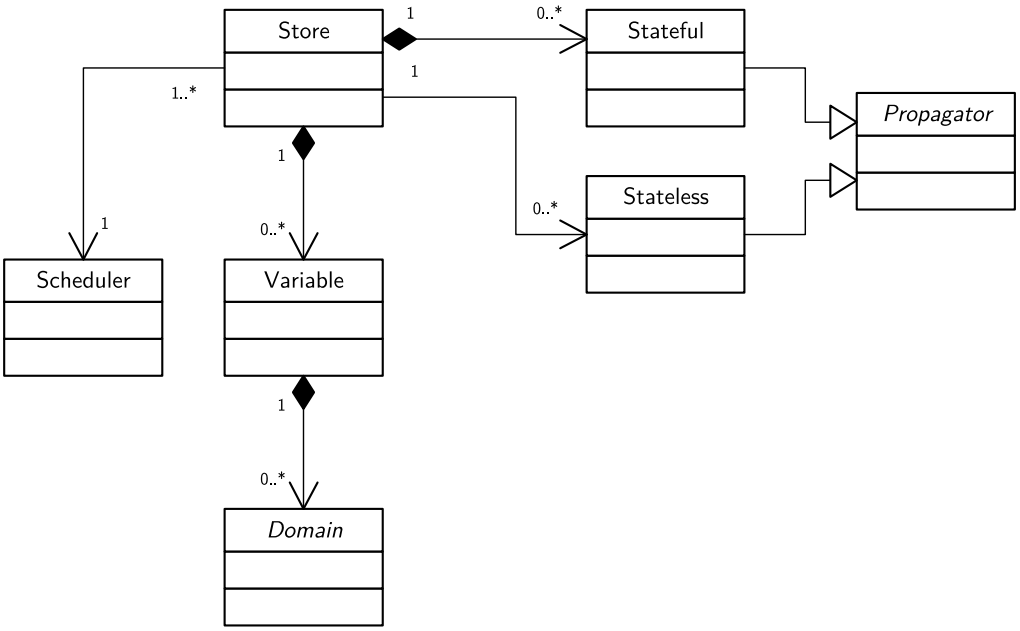


Figure 3.3.: UML model of the constraint propagation sub-system of CLFD.

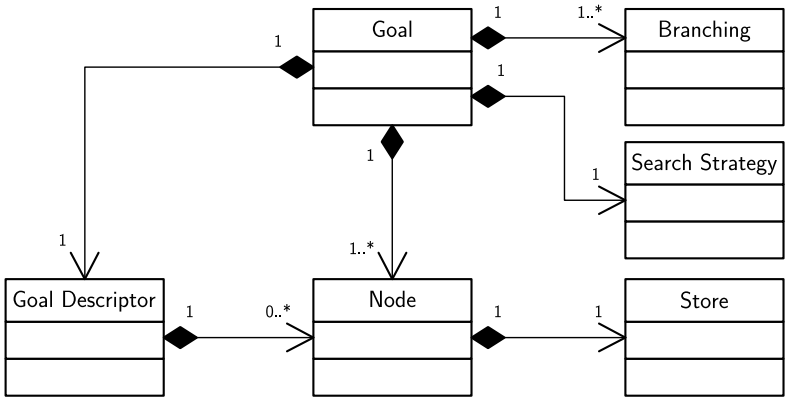


Figure 3.4.: UML model of the generic search architecture of CLFD.

This modular design comes at the price of a slight run-time overhead that the use of generic functions implies. The advantage of such a dynamic system is the possibility of a highly interactive problem construction, dynamic redefinition and exchange of modules and a sophisticated reflection mechanism (especially during interactive use). The Common Lisp Object System (CLOS) is one of the most powerful and dynamic object systems available [Bobrow *et al.*, 1987; Keene, 1989]. Unfortunately, this dynamicity does not allow the heavy inlining of function calls that statically typed programming languages and/or template based approaches such as C++ or Java permit.

However, despite its dynamic reconfiguration ability through the use of Common Lisp and CLOS, CLFD offers satisfying performance (cf. Chapter 7).

There are a number of standard protocols for adding new domain encodings, propagators, heuristics or otherwise influencing the solving process. The guiding design principle was that it should be possible to easily replace any sub-module without having to consider other system parts. This also enables a comfortable experimentation with different pruning algorithms and data structures.

3.2. Solver Design and Interface Protocols

Before illustrating how to actually describe constraint satisfaction problems and how to use CLFD to find solutions to these CSPs we will discuss the interface protocols of the individual solver parts and consider their interconnections and dependencies. Because of the modular design, almost every component can thus be replaced by a specific implementation that simply must adhere to the specified protocol. Several interface functions are provided for better performance and user convenience. We will mainly focus on the essential protocol methods the remaining protocol parts are described in the code documentation. The UML diagrams from the previous section offer a rough description of the system design. However, since it is not possible to express many of the features of CLOS by using UML, such as the meta-programming facilities or method combinations (standard or user-defined) of generic functions we have decided to additionally adapt the textural protocol representation of [Kiczales and Rivieres, 1991] to outline the inter-module contracts. The implementation details of several components will be illustrated in Section 3.3.

3.2.1. Domain Protocol

We will first analyze a most general, abstract domain component and its properties. Later we will have a detailed look at the specific domain type of integer numbers that plays a vital role in many constraint problems. The domain type is at the

very heart of every constraint problem and constraint solver. It is therefore vital to not only analyze the required functionality from an operational, but also from a performance (execution speed and space requirements) point of view. Since domain reduction is the ultimate goal of almost every solver execution step a domain representation has to find a compromise between two requirements:

- The domain modification and inspection operation should be as fast as possible since it is at the very heart of the constraint solver and thus will be one of the dominant computation activities.
- Since we will have a great number of domain instances each with a possibly high number of member values the representation should be lean and space-efficient.

All domain components should derive from a single abstract class. Due to the dynamic typing abilities of Lisp this it not strictly necessary[1], however optional type annotations are possible and enable more safety checking features in the compiler and more importantly allow for better code generation (especially with respect to CLOS method dispatch). We would therefore like to derive domains from a single abstract type/class.

The common aspects for all domain types are distilled to a very few functions. An abstract super type encapsulates some internal management data.

finite-domain-abstract [*Standard Class*]

> From this type all abstract domains are derived. It requires only a minimal set of defined operations for the actual domain implementations: A domain must be able to count the number of elements it contains, thus being able to decide if it is singleton (i.e. in solved form) or empty (i.e. the solver has failed to derive a feasible solution).

Especially the following two methods are needed to extract reflective information about the domain instance properties for heuristic, statistical purposes and to detect when a domain has been reduced to a singleton domain.

domain-contains-p (*domain element*) [*Generic Function*]

> Return the element if *element* is a member-value of *domain*. If *element* is not a member of *domain* then nil is returned.

[1] I.e. one is not required to annotate parameter and return types of functions and values.

domain-size (*domain*) [*Generic Function*]

Return the exact number of element values that are part of *domain*.

Integer Finite-Domain

Domains of integer numbers play a vital role in many constraint problems. It is therefore important to specify and analyze the required properties.

The integer domain, which ties a variable to a number of allowed integer values is the most fundamental domain. Many, if not most, constraint satisfaction problems base on this domain sub-type. Therefore it plays a central role in the CLFD implementation. Because of this fundamental part we have integrated the integer domain protocol here. Other domain types, e.g. finite set domains or graph domains will require a different interface such as different primitive pruning functions or missing min/max bounds for maps.

First, we must be able to populate a domain with allowed values.

add-range (*domain min max*) [*Generic Function*]

This function adds the range (*min max*) to *domain*. This function is primarily meant for building temporary domain representations during the solving process to be used in a difference or intersection operation.

The protocol functions in the following part are ultimately responsible for domain reduction. In the case of an empty result domain a method must signal a condition of type domain-empty (a subtype of condition). The upper layers of the solver are responsible for installing the appropriate handlers[2].

Since the domains are modified destructively for performance reasons, we do not return the reduced domain but an event describing the modification action (cf. 3.2.2).

prune-left-bound (*domain value*) [*Generic Function*]

Reduce the left boundary (minimum) value of *domain* to *value*. Note that if the element *value* is not in the domain then the resulting actual minimal value will be set to the next domain item greater than *value*.

The function is required to ensure the contracting property of propagators (cf. Chapter 4).

[2]Note that, according to the Common Lisp standard, a failure to handle a condition of this type will result in the return value nil for these functions. This is a nice property since it resembles our understanding of the according return types from a purely functional point of view.

prune-right-bound (*domain value*) [*Generic Function*]

The equivalent operation for the right-hand boundary.

remove-element (*domain element*) [*Generic Function*]

The value *element* is removed from *domain*.

domain-difference (*domain-a domain-b*) [*Generic Function*]

Compute the difference result when subtracting *domain-b* from *domain-a*. The second argument *domain-b* must not be modified in the process.

domain-intersection (*domain-a domain-b*) [*Generic Function*]

Compute the intersection of the two domains and store the result in *domain-a*. The second argument *domain-b* must not be modified during the computation.

element-intersection (*domain element*) [*Generic Function*]

Intersect *domain* with the value *element*, i.e. after application the resulting domain will have at most one element. The same operation can obviously be computed by domain-intersection with a singleton domain or by using both prune-left-bound and prune-right-bound with the same boundary value. However, this operation is at the heart of several tree search schemes and can be implemented more efficiently for many domain representations compared to the other more general operations.

Every domain type representing integer valued finite domains must provide an implementation of the above protocol methods. In Section 3.3.1 we describe the available integer domain implementations. These interface methods are employed by the pruning methods of constraint propagators to remove values from a particular domain. Since domain access dominates overall computation time in general and such methods are called many times during the solving process they should be simple and optimized.

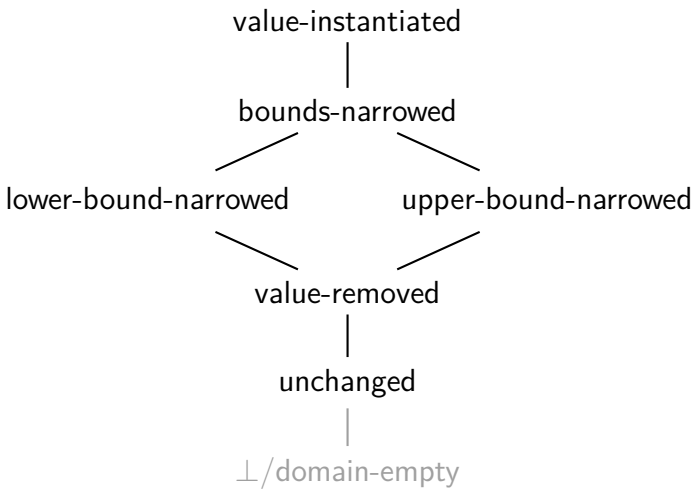


Figure 3.5.: Hasse-diagram of the domain modification event CPO.

3.2.2. Modification Events

Modification events are closely tied to the domains. The solver must be able to track and react on changes to variable domains. To enable this tracking every domain modification operation is required to return a corresponding modification event. For domains of integer sets such events are from the set $\mathcal{E}$ ={unchanged, value-removed, lower-bound-narrowed, upper-bound-narrowed, bounds-narrowed, value-instantiated}. The individual events are ordered in a way to build a complete partial order (CPO) of the form $(\mathcal{E}, \subseteq)$ as depicted in Figure 3.5.

Propagators may report a modification property. In the event of such a modification event the propagator is then scheduled for execution. The hierarchy of modification events is used to determine whether a specific propagator requires scheduling. The events are connected by the $\subseteq$-relation, i.e. an event also inhibits the scheduling of propagators tied to subordinate events. This hierarchy avoids the superfluous execution of propagators but at the same time ensures the activation of propagators which are definitely affected by a domain change. For example a propagator may be limited to narrow the bounds of integer variables. Executing it in the event of a value removal within the domain bounds is therefore unnecessary work.

The event lattice enables the scheduling of propagators to a specific variable modification type and thus reduced propagation computations. Efficient decisions about event inclusion are achieved by making use of bit-checking operations.

Ultimately, an operation resulting in an empty domain must create the event that corresponds to the bottom-element ($\bot$) of the CPO. As this case requires more

severe actions it is not implemented by using the more lightweight modification events but by employing the COMMON LISP condition system: a condition of type domain-empty must be created if any domain modification results in an empty domain. This enables us to jump to a higher level for event handling[3].

The condition is processed by a respective condition handler. If no such handler exists nil is returned by the modification function as representation of the failure event. The solver automatically sets up such handlers to manage the empty domain case at store and choice point level.

If the standard event-hierarchy is not sufficient for a type of a domain that might be added to the solver, it is possible to introduce another order of events.

3.2.3. Variable Protocol

Variables associate a name with a domain. Furthermore, the variable instances are responsible for tracking the propagator instances (i.e. constraints) a variable participates in. This enables the scheduling of only a reduced set of propagators, namely the ones tied to a modified variable. This potentially reduces execution complexity quite dramatically. Thus a variable is not only a name (as in the standard model) but a 3-tuple:

Definition 3.1 (FD-variable)**.** We define a finite domain variable $v = (\mathcal{P}_v^{sl}, \mathcal{P}_v^{sf}, D_v)$ as a 3-tuple where

- $\mathcal{P}_v^{sl}$ is the set of *stateless* propagators
- $\mathcal{P}_v^{sf}$ is the set of *stateful* propagators
- D_v is the domain of v.

The reason behind distinguishing between stateless and stateful propagators is the reduced cloning overhead during search for stateless propagators. This will be explained in detail in Chapter 4. The following operations are fundamental for the variable data type:

variable-domain (*variable*) [*Generic Function*]

> Return the associated domain instance of a variable instance. This method is implicitly defined by the underlying variable prototype class definition.

[3] An earlier implementation used a more functional approach by passing nil through several layers of method calls but it turned out that the condition-based variant not only leads to simpler code for the propagators but is also faster.

We note it here explicitly since all fundamental pruning operations (primitive constraints) are defined on domains and do not use a superfluous pass-through function as would be necessary in languages like JAVA or C++ that enforce a close class/method association for static analysis.

variable-add-propagator (*variable propagator*) [*Generic Function*]

Register a propagator with a variable, i.e. the functions inserts the propagator into to the internal managing data structure.

variable-remove-propagator (*variable propagator*) [*Generic Function*]

Remove the association of the variable with the propagator.

Both methods also dispatch on the second argument *propagator* and can therefore be implemented slightly different for both stateful and stateless propagators.

Accessing these associations is required for scheduling the propagators that are effected by a variable change. It enables the fast determination of all propagators effected by a variable modification. These propagators must in return be examined for potential execution scheduling, based on the modification type (event).

The next two methods allow the iteration through the assigned propagators.

map-variable-propagators (*variable function*) [*Generic Function*]

We iterate over all associated propagators of the variable (stateless and stateful propagators are not visited in any particular order). The function is applied to each propagator. A list of the application results is returned.

schedule-variable-propagators (*variable store*) [*Generic Function*]

> Schedule all propagators associated with *variable* if the change-event of the variable is smaller or equal (according to the event order described in Section 3.2.2) to the activation event of a propagator.

The schedule-variable-propagators method is a specialized variant of the general map iterator that schedules the associated propagators of a variable for execution.

We will see in the next section that propagators are of course able to name the variables they operate on. The previous interface methods show that variables are also able to devise a similar information the other way round. Thus the constraint network can be explored (and described) with the help of both instance types (variables and propagators).

Additionally, it is desirable to maintain information about certain variable characteristics, e.g. number of associated propagators, that may be used for solving heuristics. (These functions are not part of the fundamental interface, thus we do not list them here, but they are realized in the actual implementation.)

Variable Unification

Unifying (or aliasing) variables effectively means adding the constraint $v_1 = v_2$ for two variables v_1 and v_2 to the store thus identifying v_1 and v_2. This operation may arise as a user-defined constraint. Another variant is the inference of this constraint by a propagator. Variable unification is an important aspect to greatly enhance domain pruning throughout the constraint network and to reduce memory consumption. To enhance the possible pruning and to lower the scheduling/propagation effort for a large number of such trivial constraints it is advantageous to identify the two variables. This identification is a function of a store and supported by the implemented variable data structure.

Unifying variables not only requires their identification but also influences the additional data managed in the variable data type. Unification therefore has complex implications on both the domains and the managed propagators of the two aligned variables.

Every variable has an initially empty set of variable names it is aliased/identified with. When aliasing v_2 with a variable v_1 we execute the following operations:

- Merge the set of associated propagators, and
- Combine the domains of both variables to a unified (intersected) domain

Or more formally, let v_1 and v_2 be the following variable 3-tuples:

$$v_1 = (\mathcal{P}^{sl}_{v_1}, \mathcal{P}^{sf}_{v_1}, D_{v_1}) \qquad v_2 = (\mathcal{P}^{sl}_{v_2}, \mathcal{P}^{sf}_{v_2}, D_{v_2})$$

Aliasing v_1 and v_2 results in the following operation:

$$v_1' = (\mathcal{P}^{sl}_{v_1} \cup \mathcal{P}^{sl}_{v_2}, \mathcal{P}^{sf}_{v_1} \cup \mathcal{P}^{sf}_{v_2}, D_{v_1} \cap D_{v_2}).$$

The variable v_2 is internally replaced by a referencing data structure to v_1' such that accesses to v_2 always operate on the alias representative.

To allow this operation on arbitrary variable implementations we need to add the following method to our variable protocol.

unite-propagators (*variable-a variable-b*) [*Generic Function*]

> Merge the set of associated propagators of both variables and replace the propagator set of *variable-a* with the result set.

3.2.4. Propagation Protocol

Propagators are implementations of the pruning algorithms that implement an actual constraint. There are two main tasks a propagator must be able to handle:

- managing information about the constraint by initiating the propagator–variable connection, and
- implementing the actual pruning algorithm.

Propagators are allowed to maintain the information about the variables contributing to a propagator in any way they see fit. The attach/release functions are responsible for establishing the complementary variable–propagator connection by calling variable-add-propagator on every variable a propagator instance references.

attach-variables (*store propagator*) [*Generic Function*]

> For every variable referenced by the *propagator* the function variable-add-propagator is called to establish a mapping from a variable to an attached propagator instance.

release-variables (*store propagator*) [*Generic Function*]

Complementary to the previous function remove the established mapping by calling variable-remove-propagator for every referenced variable.

propagator-variables (*propagator*) [*Generic Function*]

Return a list of variable names (symbols) or instances the *propagator* instance is referencing.

The actual entry point to the propagator implementation is only a single method. During the propagation phase the function is called for every propagator instance of the current constraint problem.

propagate-constraint (*store propagator*) [*Generic Function*]

Execute the pruning algorithm for the respective constraint. This method either returns t or :entailed if a propagator deduces that it will always be implied from now on.

If a propagator infers that the current constellation of domains poses a conflict situation it signals a propagator-failed condition. Transitively, a domain-empty condition can be signaled by an executed primitive constraint operation, i.e. a domain reduction operation.

A propagator is free to store any additional information in its instance variables to keeps track of algorithm state or cache computation results.

3.2.5. Store Protocol

A store represents a concrete constraint satisfaction problem. It holds all information necessary to construct the relations between variables, their associated domains and the constraints defined between them. Furthermore it encapsulates a part of the heuristics assigned to a particular problem.

store [*Standard Class*]

A store describes a constraint satisfaction problem. It associates variables with their respective domains and also defines the relations between the different variables.

A scheduler heuristic is attached to a store to influence the propagator iteration process.

To describe a CSP we need methods to attach variables and their associated propagators to the store.

add-variable (*store variable*) [*Generic Function*]

> Add *variable* to *store*. The variable is an instance of type variable already with an associated domain.

add-propagator (*store propagator*) [*Generic Function*]

> Add a *propagator* instance to *store*. The instance contains references to variable names and thus represents a fully instantiated constraint relation.

To avoid unnecessary executions a solver or scheduler may decide to remove a propagator from a constraint problem entirely.

remove-propagator (*store propagator*) [*Generic Function*]

> Remove a *propagator* either by supplying an instance or only a name. This functionality is used for propagator simplification to replace a propagator by a simpler variant, e.g. when certain participating variables have been bound to particular values or have been aliased; $x = y+z \wedge y = z \Rightarrow x = 2y$.

A variable may be identified only by its name when referenced within the constraint network. This method is responsible for retrieving the variable with the correct value for the store at current search state. It is connected to the backtracking requirements of the search phase (cf. Chapter 5). The framework will initiate calls to this method automatically at the appropriate stages.

get-variable (*store name*) [*Generic Function*]

> Returns a variable, i.e. the type encapsulating a domain and associated propagators, described by a *name* symbol that is part of *store*.

Finally, a store needs a way to initiate constraint propagation and pruning.

propagate-constraints (store) [*Function*]

> This entry function starts the fix-point iteration of all propagators attached to store (cf. Algorithm 3.1). It returns either T or signal a store-failed condition.

tell-basic (store fun variable &rest args) [*Function*]

> Add (i.e. apply) a basic constraint to *store*. This primitive constraint is expressed by a relation method defined in the domain protocol on the domain type of *variable*. Apart from the obligatory *variable* the application *fun* will be completed with any number of optional arguments.
>
> If a domain will be reduced to the empty set by the constraint application a domain-empty condition is signaled, otherwise a respective modification event is returned (cf. Section 3.2.2)

Individual store implementations differ in the ways how data is managed and in the method that is used for keeping track of modifications of the managed objects (and their respective backups), e.g. copying, trailing etc. (cf. Chapter 5). All these aspects of the main managing data structure that the store represents, individually influence the overall solver performance. Several different approaches and implementation details have thus been explored.

3.3. Module Implementations

In this section we will elaborate on the details of the basic system components of domains and variables. The remaining modules, particularly propagators and stores, will be described in the following chapters.

3.3.1. Domain Representations

For integer domains several obvious implementation strategies come to mind and some have also already been exercised by constraint solvers (see also [Schulte and Carlsson, 2006]).

An obvious choice is to simply put all integers of a domain in an array of integers (CHIP compiler). The individual numbers can be easily accessed in an efficient way. However, this comes at the cost of a quite wasteful memory usage, especially for large domains.

An alternative is the encoding as a bit array where the indices represent the domain integer and a set bit at an index whether it is part of the domain (GNU Prolog, CHOCO). This reduces memory usage dramatically but at the same time maintains fast access to individual domain items.

Enumerating all possible values may prove wasteful, when a domains only contains very few elements that are wide apart on the number scale. A list of intervals

(ECLiPSE, SICStus Prolog) offers a way to reduce memory usage for such sparse domains with a large number of continuous missing numbers (so called *holes*) in a domain. Unfortunately, accessing individual items is not directly possible anymore and this scheme becomes unwieldy if a domain has a large number of holes (e.g. a domain that contains a large number of even integers only). On the other hand, all operations can be implemented in $O(m)$ (where m is the number of intervals) and in a straightforward way. For typical constraint problems where the domains are not extraordinarily large this approach scales quite well and is also easily implemented in higher order languages such as Prolog. Unfortunately, it performs quite poorly for typical benchmark problems such as n-Queen (for large n, cf. Section 7.2.2) because they result in domains with many holes and widely distributed single values.[4]

A related approach with similar drawbacks is employed by Screamer which uses an integer domain encoded by its lower and upper bounds and additionally keeps a list of already removed values. Compared to the previous approach this simplifies operations such as domain difference and union.

To cushion the linear scan of the interval list when inner domain values are accessed (either for removal or inclusion testing) a data structure of a self-balancing tree of intervals could be useful but to our knowledge has not yet been implemented by a solver. However, complex operations such as domain intersection are quite complex to implement (cf. below).

If a solver is limited to operations and reductions on domain bounds it is of course sufficient to encode a domain simply by its boundary values. Since such a solver does not prune any inner domain values it generally results in large search trees for typical CSPs.

Apparently a list of intervals has been a favored choice. This data structure has many advantages and is also one alternative representation in CLFD.

The available domain representations each provide a trade-off between efficiency and space requirements. This results also in a difference in the representable value ranges. Figure 3.6 illustrates the hierarchy of domain implementation that support integer elements as present in CLFD. All domain types inherit from an abstract type for all domains. The domain of integer numbers additionally is derived from an abstract integer-domain. All generic functions of the integer domain protocol (cf. Section 3.2.1) are defined on this type. The next paragraphs illustrate the

[4] Hash-tables may present a viable alternative as the underlying data structure for sparse domains. However, their internal representation is far too heavy for a domain data type where easily several thousand instances may exist at one moment. Furthermore, hash-table iteration does not guarantee the correct numeric order for the domain values, which complicates algorithms and requires temporary data structure creation and additional sorting steps thereby degrading overall performance.

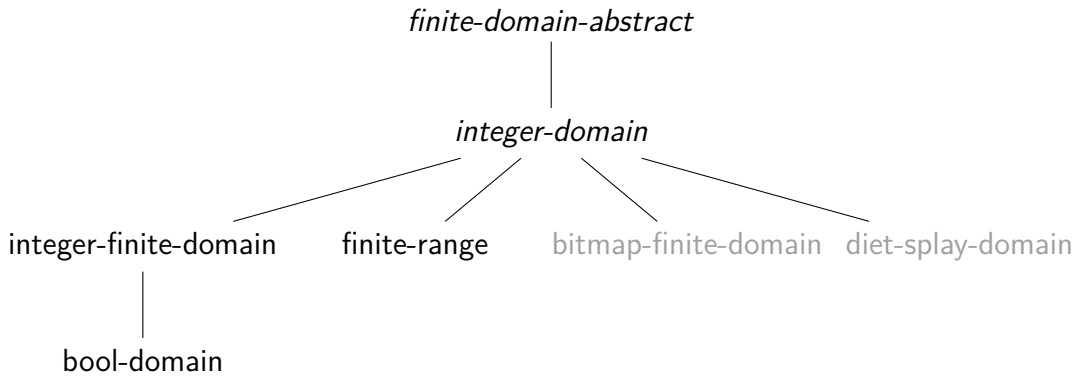


Figure 3.6.: Hierarchy the finite-domain data type on integers.

individual implementations.

Generally integers of unlimited size[5] are supported by the solver, however, some domain implementations restrict the allowed minimal and maximal range due to data structure constraints. All domain implementations allow holes in their representation, i.e. values can also be removed from the middle between two range bounds. This is important because it enables much better pruning and consequently much reduced search complexity, compared to a bounds-only capable solver.

Integer based Finite Domain (integer-finite-domain)

This domain representation uses the standard Common Lisp integer number type. Integers are limited in their size by the machine's available memory only. We use the individual bits of such an integer number to indicate if the number described by the index of a bit is part of a domain (the bit is set to 1) or not (the bit is 0). The current implementation of this domain type only uses one integer for the set which means that the lowest representable value of a domain is 0. It would be simple to allow values below zero by introducing a second integer for the values below 0 that belong to the domain.

This representation also has the advantage that the compiler can automatically use the CPU's bit operations on the 32/64 bit wide blocks of the integer number to perform most of the domain operations yielding much better performance compared to accessing individual bits. Furthermore the (memory) size used for the integer will scale automatically depending on the largest index bit set.

Example 3.2. The encoding of a set of numbers by the corresponding bit indices. Here, these bits represent the integer 570. Remember that individual bits are

[5] i.e. only limited by the available memory in a system

enumerated from the right, so the number 570 resembles the bit-wise representation of the domain $\{1, 3\text{–}5, 9\}$.

$$\{1, 3\text{–}5, 9\} \Rightarrow 570 \Rightarrow 0\text{x}b\underbrace{1000111010}_{\text{dynamic size}}$$

An additional benefit is, that the number of required bits scales dynamically (and automatically) in correspondence to the highest domain value.

Range-List based Finite Domain (finite-range)

Here, the set of values is represented as a list of ranges, i.e. a list of pairs where each pair represents sequence of numbers without any holes.

Example 3.3. Using the same set as in Example 3.2 the implementation leads to the following range list:

$$\{1, 3\text{–}5, 9\} \Rightarrow ((1, 1), (3, 5), (9, 9))$$

Obviously, access to individual set values is algorithmically slower (and implementationally more complex) compared to the integer based representation. However, we found that performance is acceptable especially since typically domains do not contain many little holes such that the range list becomes problematically long. This might change with the addition of further pruning approaches.

Boolean Finite Domain (bool-domain)

The Boolean domain is a sub-type of the integer finite domain. The allowed elements of the set $\{false, true\}$ are represented as $\{0, 1\}$ as this eases the modeling of many constraint problems [Smith, 2005; Artiouchine *et al.*, 2005].

Deprecated Finite Domain Representations

We have experimented with two other domain representations for integer valued domains. The first one was a domain based on bit-vectors a second representation was based on a tree data structure.

Before settling for the integer number domain representation based on memory-limited integers as described above, we used the simpler approach of employing a *bit-vector* where the bit value at each index indicated whether the index-value was part of the domain value set or not. However, to use the (efficient) built-in functions for AND-ing or OR-ing two vectors, the parameter vectors must be of equal length. Therefore the vectors had to be set to a compile-time constant value which

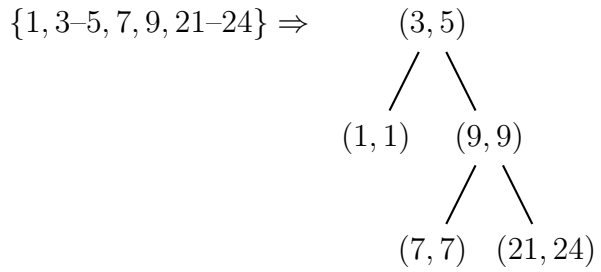


Figure 3.7.: Discrete interval encoding tree representation of a set of integers.

defeats our goal of an efficient set representation. Additionally more processor time is needed for copying potentially unused parts of the vector since their size does not scale dynamically. The more low-level integer based representation overcomes these limitations.

Another approach represents the set of integer values in a tree of ranges (discrete interval encoding tree [Erwig, 1998]). An example illustration is presented in Figure 3.7.

In contrast to Erwig we used a self-balancing splay tree [Sleator and Tarjan, 1985] to obtain a self-balancing data structure for the range nodes. However, the more complex domain operations, such as intersection and difference, were quite intriguing to implement. Furthermore, tests revealed that there was no speed advantage compared to the simpler range-list approach that would warrant the higher code maintenance costs. Apparently, the range-lists are short enough, that any advantage of the tree structure is leveled by the necessary balancing/splay operations.

The deficiencies of these two domain representations were too grave such that they were abandoned.

3.3.2. Variables

Variables are container objects that map a name to a domain (cf. Definition 3.1). Furthermore a variable instance is responsible for tracking the constraints a variable v participates in by managing the propagator instances v is referenced from.

The major operations are:

- to add or remove a propagator–variable connection, and
- to iterate over all associated propagators to schedule them for activation,

While their simplicity does not make it worthwhile to experiment with different variable implementations, the modular architecture makes this nevertheless possible.

The current variable representation simply uses two lists to track the stateless and stateful propagators. Since new propagator instances are only added to the beginning of a list we can share the list tails with different (cloned) variable copies. Iteration is obviously also trivial and efficient.

The only operation having a drawback is propagator removal, which is now of $O(n)$ complexity (where n is the length of the list) because the search for a specific propagator requires a linear scan of the list. This operation only happens when a propagator instance was entailed. Since this is a fairly infrequent event compared to the other operations there was no need (yet) for more elaborate data structures.

Additionally, a variable instance will be used for guiding the search heuristics when statistical information such as the number of associated propagators is accessed. It is worthwhile to add a few statistical counters (e.g. number of assigned propagator instances) to enable an efficient heuristic analysis. These values can be queried through interface functions and processed for heuristic guidance (cf. Section 5.2.3).

Users are of course free to derive their own specific implementations with added management and statistical information.

4. Propagation

Propagation is at the heart of our constraint solving engine. This chapter describes the notions, requirements and implementation details of the propagation machinery. Section 4.1 introduces basic local consistency iteration and details the various parameters that influence this process. Next, Section 4.2 describes a number of important propagation algorithms that have been implemented and advanced within CLFD. Sections 4.3 and 4.4 illustrate techniques which enable the development of complex propagators and support the efficient update of complex internal graph data structures. Finally, Section 4.5 highlights related work that has been done within the context of the CLFD architecture.

The propagation phase is responsible for applying the associated constraints of a constraint system. Such constraints are represented as instances of the propagator class. Every instance references a number of variables and values that form a particular constraint expression.

Definition 4.1 (propagator)**.** A *propagator* p implements a domain reduction operation. It consists of two parts: a class (derived from the abstract propagator class) for managing variables/domain values and internal data structures, and a accompanying propagate-constraint method that performs the actual reduction. (Additional management functions for the propagator protocol are discussed in Section 3.2.4.)

In our framework we distinguish two general variants of propagator types: *stateless* and *stateful* propagators. This discrimination is primarily introduced for performance and memory reduction reasons. The two propagator types have the following properties:

stateless propagators do not carry an internal state but implement a fully functional data type. Therefore there is no need to initiate copying operations for such instances during a (backtracking) search phase.

stateful propagators may reference an internal state. The programmer must provide a respective clone operation that is used for backup retrieval of such data structures.

Since the backup operations require memory access and additional space it is advantageous to execute them only for the necessary propagators. This is a very important optimization since it greatly reduces the number of function calls for the cloning phase, reduces overall memory consumption and eases debugging. Overall, stateless propagators can be handled much more efficiently. Many propagators, especially for the smaller arithmetic constraints fall into this category. Stateful propagators, on the other hand, are important for global constraints where usually a complex graph based data structures is constructed and incrementally modified during propagation.

To ensure proper computation and termination (cf. Chapter 6) propagators are required to fulfill the following properties:

monotonicity ensures that the order of two elements is not changed by the application of pruning function ($x \leq y \Rightarrow f(x) \leq f(y)$). This property guarantees that a result domain is not smaller after the pruning application than another comparable domain which had fewer values before the applications.

inflation with respect to the underlying domain type, i.e. all domain changes must either result in a smaller result domain or in no change at all. To ensure $x \preceq f(x)$ it is forbidden to add any values to the domain.

These properties are exploited during the propagation phase where the constraint system is evaluated to reach an arc- or hyperarc-consistent state (cf. Section 2.3). We will elaborate on this process in the following section. Chapter 6 highlights and employs the required properties from a theoretical point of view by analyzing the termination requirements of the solver system.

In Section 2.3 we have established several consistency notions to describe solutions of a CSP. A propagator is an algorithm to remove inconsistent values from variable domains. Since so far we have only considered enumerable domains, our consistency notions were defined on every item.

Propagators that consider all domain values for consistency are said to be *domain consistent*. This is equivalent to the notion of hyper-arc consistency of Definition 2.14.

Often, the explicit consideration of every domain value is prohibitive due to the required computational complexity. Therefore, some pruning algorithms only consider the lower and upper bound of a domain for their pruning and consistency test. A constraint C is therefore consistent of the lower and upper bound of a domain participate in a solution to C. This provides weaker pruning but at the same time is computationally much simpler. Such a propagator is said to enforce *bounds consistency* [Apt, 2003].

Definition 4.2 (bounds consistency)**.** Consider a CSP $C = C' \wedge x_1 \in [min_1, max_1] \wedge \ldots \wedge x_n \in [min_n, max_n]$ with $C' = c_1 \wedge \ldots \wedge c_k$.

A constraint c with $var(c) = \{y_1, \ldots, y_m\} \subseteq \{x_1, \ldots, x_n\}$ of the conjunction C' is *bounds consistent* if $\forall y \in var(c)$ with $y \in [min, max]$ there exist $d, e \in C$[1] such that $l_y = d[y]$ and $h_y = e[y]$.

A CSP C is *bounds consistent*, if all $c_1, \ldots, c_k$ of C' are bounds consistent.

Several constraints supported by CLFD are based on bounds consistent pruning or at least provide are variant with a bounds consistent pruning algorithm, e.g. the *all-different* propagator that is provided in a variety of different pruning strengths.

4.1. Consistency Iteration

The result of the propagation phase is an arc-consistent constraint system (or hyperarc-consistent, depending on the employed propagators), i.e. with the current constraint instances it is not possible to remove any more values from the variable domains. The standard algorithm in wide use to reach this fix-point is the AC3-algorithm [Wallace, 1993]. A pseudo-code display of AC3 is presented in Algorithm 4.1.

We start with a set (or queue) Q of propagators. Every propagator instance that is added to a constraint system (store) is initially scheduled for propagation. Next, one propagator is removed from the scheduled set and its pruning method propagate-constraint is invoked (line 2–3). This will result in either no changes, or (ideally) in a number of variable domain reductions (line 4). The change events of these domain reductions are collected and a set of changed variables $v_{changed}$ is returned. Finally, all propagators associated with the affected variables are joined with the scheduled propagator set Q (line 5–7). The algorithm finishes when this scheduled propagator set becomes empty (line 1).
During propagation the solver may deduce that the current constraint system as represented by the store is not consistent. This is the case if a variable is reduced to the empty domain. As a second possiblity, a propagator may infer inconsistency before executing any domain reductions. In both cases a global failure is signaled. These signals are handled by the catch phrase in line 9.

Idempotence of a propagator is not a required property. But it is an important property for optimization purposes.

[1] Recall Definition 2.6 where we defined constraints as a subset of $D_1 \times \cdots \times D_n$. Here, we take advantage of this definition and directly access the domain bindings for the two constraints d and e at place y.

Algorithm 4.1: Basic AC-3 arc-consistency algorithm.

Input: A store S.
Result: The arc-consistent store S.

Data: Q is the set of scheduled propagators.

```
1  while Q is not empty do
2      p ← arbitrary element of Q;
3      try V_changed ←propagate-constraint(p) on S
4          foreach v ∈ V_changed do
5              foreach propagator p_v associated to v do
6                  {p_v} ∪ Q;
7              end
8          end
9      catch domain-empty ∨ propagator-failed
10         return nil;
11 end
12 return t;
```

idempotence Propagators with an idempotent execution property ensure that $p(p(X)) = p(X)$ holds, i.e. an immediate second application of propagator p on a set X of variables must not result in any further changes. This results in propagators computing their individual fix-points as a result of their execution.

If a propagator is known to have reached its fix-point after execution, any further run would not gain an additional domain reduction. A scheduler is thus free not to re-schedule the current propagator if it is know to be idempotent. Line 5 may then be changed to $p_v \neq p$.

All schedulers that use the automatic re-scheduling interface of our framework automatically take advantage of propagator idempotence and do not reschedule the current propagator instance for another run.

It is the responsibility of the implementer to announce the non-idempotence of a propagator. This is done by including the `non-idempotent-mixin` in the inheritance list of a propagator. In CLFD idempotence is assumed as the default since this is the case for almost all implemented propagators. Furthermore, most global constraint pruning algorithms inherently ensure this property.

Note, that this mixin property may also be dynamically removed due to the dynamic properties of the underlying CLOS should the need arrive, e.g. when

dynamically selecting different pruning algorithms for a particular propagator, depending on the current store state.

4.1.1. Propagator Scheduling

Every constraint system has an associated scheduler that is responsible for managing the propagation phase. There is exactly one scheduler instance per constraint problem[2]. Basically it implements two tasks:

- management of the set of scheduled propagator instances, and
- consistency iteration over the necessary propagators.

A store may choose to duplicate its associated scheduler. This action is only necessary when we strive for a concurrent evaluation (cf. Section 9.1.1).

The scheduler is one of a number of major aspects contributing to a solving heuristic and thus a main contributor to the overall performance. Primarily it is responsible for picking the next propagator from a set of scheduled propagator instances and initiate its execution. Secondly, it implements one of the many arc-consistency schemes established in the field of constraint propagation (cf. below).

Figure 4.1 displays a simple scheduler implementing the AC-3 consistency algorithm. The depicted scheduler implementation uses a FIFO-queue. After the execution of a propagator the other propagators effected by the induced changes of a variable domain are therefore appended to the queue list and thus tend to be run only after some time. Another viewpoint might be that such derived effects should be propagated as soon as possible. One could easily replace the queue implementation with a simple LIFO stack to obtain the desired effect.

A scheduler is responsible for the arc-consistency iteration. Of course schedulers are not limited to the above AC-3 algorithm but are free to implement stronger or weaker consistency algorithms like AC-4 or the recently proposed AC-3.1 or AC-2000 [Bessière *et al.*, 2005]. However, most constraint solvers use a form of AC-3 consistency as it represents the best compromise of computation complexity, domain pruning effectiveness and memory requirements [Dechter, 2003; Wallace, 1993].

Apart from the consistency iteration an important part of a scheduler is the selection of the propagator that is going to be executed at the next iteration step. A scheduler mainly can base its decision on one or a combination of the following aspects:

[2]For a parallel execution scheme the scheduler must be duplicated such that every running instance can manage its internal state. This application case is supported in the method protocol.

```
(defclass basic-scheduler (propagation-scheduler)
  ((queue :initform (make-instance 'fifo-queue)
          :accessor scheduler-queue)))

(defmethod run-propagation ((scheduler basic-scheduler) store)
  (let ((queue (scheduler-queue scheduler)))
    (loop for propagator = (dequeue queue)
          while propagator do
            (propagate-and-schedule store propagator)
          finally (return t))))
```

Figure 4.1.: Implementation of (a very simple variant of) the AC3-Algorithm.

1. the time or order of the scheduling request of a specific propagator,
2. the (assigned) priority of a (set of) propagators,
3. the computational complexity of a propagator,
4. the possible impact of a propagator application [Ringwelski and Hoche, 2005].

Due to the modular architecture we are able to employ a fitting scheduler on a specific problem class. For example the `5-stage-scheduler` makes use of the complexity classification associated with each propagator implementation. This specific scheduler prioritizes the less expensive propagator instances of a CSP over the more costly ones. The reasoning behind this heuristic is that it is advantageous the remove as many values as possible without much computational effort making the application of the more sophisticated pruning algorithms even more effective. Scheduling based on an assigned propagator priority can be realized by using a priority queue for the scheduling data structure. In [Schulte and Stuckey, 2004] it is shown that such techniques can lead to a dramatic solving speed-up and search space reduction.

Currently CLFD distinguishes five different propagator complexities similar to the proposal in [Schulte and Stuckey, 2004], ordered from simplest and fastest to slowest:

1. *simple* (constant) complexity. Propagators of this type are not scheduled but applied immediately.
2. *linear* complexity,

3. *sub-quadratic* complexity,
4. *quadratic* complexity,
5. *complex* ($n > 2$-polynomial or exponential) complexity.

Each propagator implementation is a member of one of these complexity types. The complexity is a property of its type, i.e. the information is exploitable by the compiler to generate an efficient method dispatching scheme. As with all type aspects, a propagator may change its complexity property at run-time. An example would be the *all-different* propagator which may choose to use a graph-based pruning approach at an early propagation stage and then switch to a simpler method that only reacts on domain instantiations once an initial domain reduction has been achieved.

Currently, CLFD provides the following schedulers:

FIFO is a scheduler where the propagators are entered in a queue, i.e. a propagator that is enqueued before another propagator will also be executed first.

LIFO is similar but with a stack-like behavior.

5-stage employs a priority queue to order propagator execution based on computation complexity (hence the name for the five complexity variants) and propagator priority.

Additionally, a propagator may alter its default priority if it decides that it is necessary to be re-run as soon as possible. This makes it possible to easily align a number of interdependent propagators that require a particular order of execution for correct results.

The scheduler uses a pairing heap [Fredman *et al.*, 1986] to handle the differently prioritized propagators[3].

Integrating a new scheduler into the framework is simple. The framework provides a controlling instance that transparently manages repeated scheduling (i.e. tracking of already scheduled propagators) and idempotence handling. The scheduler only needs to supply the two methods for inserting and removing a propagator to/from a scheduler queue and an implementation of the internal scheduler data structure.

The accompanying algorithm for consistency iteration is optionally exchangeable.

[3]We experimented with a number of other priority queue data structures, e.g. Fibonacci heaps, but their performance was inferior in comparison, especially with respect to the handling of many equally prioritized queue items.

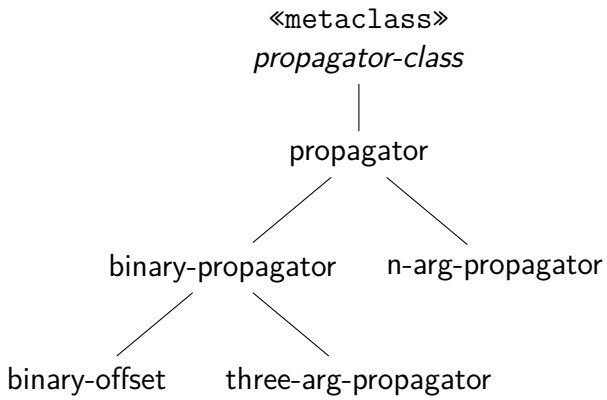


Figure 4.2.: Hierarchy of fundamental (abstract) propagator types.

4.2. Propagators

Propagators are at the heart of the domain pruning phase as they implement the algorithms that actually infer the conflicting values of a domain. CLFD includes a number of constraint algorithms. All propagator functions described in this section operate on integer set domains. The pruning operators infer all non-solution values of a domain and initiate a sequence of basic constraints (cf. Section 3.2.1) that execute the actual domain reduction.

We have classified the currently existing propagators by their respective (code-) complexity and algorithmic methods and separated their description in the following sections[4].

4.2.1. Simple Propagators

A number of base propagators have been implemented during the development to test the overall design in the early stages. Some of these often simple constraint cases will be required as base operations in the next section, where we add the ability to process more complex expressions. Pruning for arithmetic expressions is usually implemented by bounds-consistency only. This is also the case for our arithmetic propagators. All propagators are derived from one of the basic propagator classes that are displayed in Figure 4.2. These base classes already provide basic management facilities for variables, caching and scheduling.

We provide propagators for the following arithmetic relations:

[4]This is only a separation to ease and cluster the propagator descriptions and has only a superficial relation to the complexity classification as employed by the solver framework as described in Section 4.1.1.

$x\ op\ y + c$ where $op \in \{=, \neq, \leq, \geq, <, >\}$ implements a binary relation between two variables and an integer-valued offset.

$x = y + z$ a ternary polynomial relation. The pruning can be trivially (and idempotently) calculated in one pass.

$x = y \cdot z$ another ternary polynomial that serves as one of the base cases for the handling of non-linear polynomials. The relation $x = \frac{y}{z}$ can be expressed by this propagator as well by applying a simple arithmetic transformation.

$x = y^n$ the exponentiation relation is also required for the handling of non-linear expressions (cf. Section 4.2.2).

A better Algorithm for Interval Integer Division

The propagators for $x = y \cdot z$ and $x = \frac{y}{z}$ on integer variables require an operation for division on integer intervals of two variables, i.e. an implementation of the v/w operation for two variables with integer domains. In accordance to [Apt, 2003], the division operation is defined as

$$X/Y := \{u \in \mathbb{Z} | \exists x \in X, \exists y \in Y \text{ such that } u \cdot y = x\}$$

The difficulty of this operation is that the result may not be an integer and that the operation cannot be expressed by simple operations on the interval bounds. It is not straightforward to express this situation as integer interval. In [Apt and Zoeteweij, 2004] a detailed analysis of integer interval operations is given. We start by outlining the proposed method by Apt *et al.*

Consider the operation $int([a..b]/[c..d])$ on integer intervals where *int* indicates that the result should be coerced to an integer interval. It is defined as

$$int(x) = \begin{cases} \text{smallest integer interval containing } x & x \text{ is finite} \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

For the interval division we distinguish five cases:

Case 1: $0 \in [a..b]$ and $0 \in [c..d]$ then $int([a..b]/[c..d]) = \mathbb{Z}$ by definition of *int*.

Case 2: $0 \notin [a..b]$ and $c = d = 0$ then $int([a..b]/[c..d]) = [] = \emptyset$

Case 3: $0 \notin [a..b]$ and $c < 0$ and $d < 0$ then $int([a..b]/[c..d]) = [-e..e]$ where $e = \max(|a|, |b|)$

Case 4: $0 \notin [a..b]$ and either $c = 0$ and $d \neq 0$ or $c \neq 0$ and $d = 0$ then $int([a..b]/[c..d]) = int([a..b]/([c..d] - \{0\}))$

Case 5: $0 \notin [c..d]$.

It is an initial observation when dividing the bounds that the following relation holds: $int([a..b]/[c..d]) \subseteq [\lceil \min(A) \rceil .. \lfloor \max(A) \rfloor]$ where $A = \{a/c, a/d, b/c, b/d\}$.

The problem is that equality may not hold for this case. Consider the example $int([155..161]/[9..11]) = [16..16]$. But with $A = \{155/9, 155/11, 161/9, 161/11\}$ the resulting interval is $[\lceil \min(A) \rceil .. \lfloor \max(A) \rfloor] = [15..17]$ thus resulting only in a $\subseteq$ relation. This deviation occurs, because 16 is computed by 160/10, but both numbers, 160 and 10, are not the boundary values of the expression intervals.

To ensure equality Apt *et al.* propose to pre-process $[c..d]$ such that the bounds are actual divisors of $[a..b]$. We look for the smallest $c' \in [c..d]$ such that $\exists x \in [a..b], \exists u \in \mathbb{Z}$ where $u \cdot c' = x$. This requires a *linear search* for such a c'. Similarly, we look for the largest $d' \in [c..d]$ which fulfills analogous conditions. The sought equality now holds with

$$int([a..b]/[c..d]) = [\lceil \min(A) \rceil .. \lfloor \max(A) \rfloor] \text{ where } A = \{a/c', a/d', b/c', b/d'\}.$$

The linear search required in *Case 5* can be quite costly and noticeable for large numbers. We therefore propose a different scheme for obtaining c' and d' that computes the new bounds with significantly fewer steps (but may escalate to near linear behavior in borderline cases). Algorithm 4.2 demonstrates the computation for c', the computation for d' is equivalent.

Theorem 4.1 (Equivalence of computed solutions)**.** Both algorithms, linear search and our quick-stepping alternative, compute the same interval.

Proof. Line 4 computes c' such that it is the smallest integer in $[c..d]$ with $c' \cdot \lfloor b/c \rfloor \overset{?}{\in} [a..b]$. The predicate $c' > d$ in Line 5 tests, whether the computed c' falls outside the original interval, in that case there is no actual divisor in within the interval bounds and a failure is signaled. Furthermore, for every $c'' < c'$ holds $c'' \cdot \lfloor b/c \rfloor < a$ and thus, $\forall e \leq \lfloor b/c \rfloor$ there holds as well $c'' \cdot e < a$, and thus c'' cannot have a multiple in $[a..b]$.

In Line 8 we compare with the previously computed bound. If they match, our smallest fix-point is reached. From $c' = c = \lceil a/\lfloor b/c \rfloor \rceil$ follows $c \cdot \lfloor b/c \rfloor \geq a$ which is the required property. Thus c' is the smallest number in $[c..d]$ with this property because for $\forall c'' < c'$ holds $c'' \cdot \lfloor b/c \rfloor < a$, and thus $\forall e \leq \lfloor b/c \rfloor$ there holds $c'' \cdot e < a$ as well, and c'' cannot have a multiple in $[a..b]$. □

Algorithm 4.2: Algorithm for quick-stepping computation of lower actual divisor bound c'.

Input: interval borders a, b, c, d.
Result: lower actual divisor border c'
`// initialize` c'
1 $c' := c$;
2 **repeat**
3 $\quad c := c'$; `// save last step for fix-point test`
4 $\quad c' := \lceil a/\lfloor b/c \rfloor \rceil$;
5 $\quad$ **if** $c' > d$ **then**
$\quad\quad$ `// Return empty interval`
6 $\quad\quad$ return nil;
7 $\quad$ **end**
8 **until** $c' = c$;
9 return c';

In [Apt and Zoeteweij, 2006] it is suggested to use the relaxed method of *weak division* (denoted as :) to compute the bounds which avoids any candidate search completely. It introduces a new rule for interval computation of the result bounds:

$$[a..b] : [c..d] := \begin{cases} [\lceil \min(A) \rceil .. \lfloor \max(A) \rfloor] & 0 \notin [c..d] \text{ or,} \\ & 0 \notin [a..b] \text{ and } 0 \in [c..d] \text{ and } c < d \\ [a..b]/[c..d] & \text{otherwise} \end{cases}$$

where $A = \{a/c', a/d', b/c', b/d'\}$, and $[c'..d'] = [c..d] - \{0\}$.

We have found our method to yield significantly smaller intervals than weak division but at the same time require much fewer steps for the candidate search than a simple linear scan.

Example 4.1 (Divisor search vs. weak division)**.** Consider the following interval division [1100001..1100001]/[8000..120000]. Our method computes [21..49] as result interval. The result for weak division is [10..137], which is significantly wider.

4.2.2. Polynomial Expressions

Polynomial expressions build the foundation for the description of many constraint problems. In CLFD we base such expressions on a number of other constraint propagators. Linear polynomial expressions of n variables are handled by a corresponding propagator. The management of linear integer expressions was extensively analyzed in [Apt and Zoeteweij, 2004; Zoeteweij, 2005].

Basicly a linear expression is of the form

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \ \ op \ b \text{ where } op \in \{=, \leq, \geq\}$$

Since we have to handle positive and negative coefficients differently we identify them by their respective index that is assigned to a set *NEG* or *POS* representing positive and negative integers according to the original coefficient. The a_i are therefore positive integers only. All the x_i must be distinct variables. The right-hand side b is an integer number as well.

In accordance to [Apt and Zoeteweij, 2004] the pruning rules for the equality relation can be expressed as a combination of the rules for the $\leq$ and $\geq$ relations which we will regard first. Depending on whether $i \in POS$ or $i \in NEG$ we are able to prune the lower or upper bound of x_i. The following rules effectively express a repeated transformation of the relation for every i such that $1x_i$ is the polynomial on the left-hand side and thus the bounds for x_i can be deduced based on the other variable domains by direct computation.

$$\sum_{i \in POS} a_i x_i - \sum_{i \in NEG} a_i x_i \leq b$$

For every $x_j = (l_j, h_j)$ we consider two cases (cf. [Apt and Zoeteweij, 2004]):

$j \in POS$**:**

$$\alpha_j := \frac{b - \sum_{i \in POS - \{j\}} a_i l_i + \sum_{i \in NEG} a_i h_i}{a_j}, \ x_j \in (l_j, min(h_j, \lfloor \alpha_j \rfloor))$$

$j \in NEG$**:**

$$\beta_j := \frac{-b + \sum_{i \in POS} a_i l_i - \sum_{i \in NEG - \{j\}} a_i h_i}{a_j}, \ x_j \in (max(l_j, \lceil \beta_j \rceil), h_j)$$

Similar rules are inferred for the corresponding $\geq$-Relation.

$$\sum_{i \in NEG} a_i x_i - \sum_{i \in POS} a_i x_i \leq -b$$

Again, for every $x_j = (l_j, h_j)$ we consider two cases:

$j \in POS$**:**

$$\gamma_j := \frac{b - \sum_{i \in POS - \{j\}} a_i h_i + \sum_{i \in NEG} a_i l_i}{a_j}, \ x_j \in (max(l_j, \lceil \gamma_j \rceil), h_j)$$

$j \in NEG$**:**

$$\delta_j := \frac{-b + \sum_{i \in POS} a_i h_i - \sum_{i \in NEG - \{j\}} a_i l_i}{a_j},\ x_j \in (l_j, min(h_j, \lfloor \delta_j \rfloor))$$

Put together we can infer for the equality relation that for every $x_j = (l_j, h_j)$ the two cases are:

$j \in POS$**:**

$$x'_j := (max(l_j, \lceil \gamma_j \rceil), min(h_j, \lfloor \alpha_j \rfloor))$$

$j \in NEG$**:**

$$x'_j := (max(l_j, \lceil \beta_j \rceil), min(h_j, \lfloor \delta_j \rfloor))$$

Generally we are interested in idempotent propagators. One iteration of the pruning rule through all variables is obviously not guaranteed to be idempotent since the pruning of a variable x_j may lead to further possible pruning in a dependent variable x_i $(i \neq j)$ in the same relation. Therefore, to obtain an idempotent propagator the application of the reduction rule to all variables of the relation must be repeated until no further domain reductions occur.

Furthermore, to apply the reduction rules the polynomial expression must be linear and all variables must occur only once. To ensure the latter condition for arbitrary polynomials entered by the user and as a stepping stone to handle non-linear polynomials we transform and maintain the input expression into a canonical form.

Canonical Forms and Handling of Non-linear Expressions

A unique feature of CLFD is its automatic simplification and splitting of complex arithmetic expressions to derive an efficient representation of an expression by using the most efficient propagators for an expression. At the same time this step dramatically reduces the number of required propagators. A smaller number of propagators generally results in a much more efficient propagation process.

Canonical form simplification has been used by algebraic manipulation programs for a long time [Martin and Moses, 1970]. The idea is to translate any two equal expressions into an identical (internal) canonical form on which further manipulations are exercised. In [Norvig, 1992] Norvig discusses a canonical form for polynomials which is then used to form a fully functional simplifier for non-rational polynomials. The simplifier in CLFD is an expanded variation which handles a broader range of arithmetic operations.

For our purposes we can define three different types of input polynomials (in adaption to Norvig):

1. numbers (rationals in the implementation) are polynomials,
2. constraint variables, i.e. variable names represented as Lisp symbols, are polynomials,
3. if p and q are polynomials then $p + q$, $p * q$ and p/q are polynomials as well (subtraction is expressed by polynomial addition and sign-inverted coefficients, i.e. multiplication by -1, for the subtracting polynomial)

Polynomials have a *main variable*, *coefficients* and a *degree* (the highest power). For example $ax^2 + bx + c$ is a polynomial with the main variable x, degree 2 and the coefficients *a, b, c* of type `integer`. Internally the polynomials are represented as a vector[5] where the main variable is at position (i.e. index) 0 and further positions are occupied by the coefficients for x^0 ... x^n respectively, where x is the main variable. Numbers are represented as themselves. Thus we have the following internal realizations:

$x^2 - 2x + 5$	is represented as the vector `#(x 5 -2 1)`
x	is represented as `#(x 0 1)`
5	is represented as `5`
p/q	is represented as a rational expression with numerator p and denominator q which themselves are again polynomials.

Coefficients are additional polynomials, either simple numbers or polynomial representing vectors. To get a final canonical form we need to impose an ordering on the variables to decide about the main variable of expressions like $(x + y)$. This ordering is defined as the alphabetical (lexicographic) ordering on variable names.

Thus a polynomial $x + y$ can only be represented as `#(x #(y 0 1) 1)` and not `#(y #(x 0 1) 1)`, i.e. a polynomial p can only have coefficients of polynomials in variables greater (according to the lexicographical ordering) than the main variable of p, not smaller.

The arithmetic operations are realized by the usual rules for polynomial arithmetic. A special case are rational expressions (i.e. polynomial divisions). Such rational expressions are always computed to their most simplified unique form. i.e. for a polynomial $\frac{p}{q}$ the numerator p and denominator q are shortened by dividing them by their (polynomial) greatest common divisor.

After converting an input polynomial relation expression from an external representation to its internal form it is a simple matter of polynomial arithmetic to collect the variables of both relations sides in one polynomial. Possible duplicate appearances of variables are automatically collected.

[5] A vector is a one-dimensional array.

Example 4.2 (Expression simplification)**.** The simplifier is able to infer much simpler forms for complex expressions. Consider the expression[6]

$$e = \frac{\frac{k(k-1)(k-2)(k-3)(k-4)}{120} - \left(\begin{array}{l} (\frac{k(k-1)}{k-2} - 720)\frac{(k-4)(k-3)}{2} \\ +\frac{k}{12}\frac{(k-4)(k-3)(k-2)(k-1)}{24} \\ +\frac{1}{2}\frac{k(k-1)(k-2)(k-3)(k-4)}{120} \\ +\frac{1}{k}\frac{k(k-1)(k-2)(k-3)(k-4)}{720} \end{array} \right)}{k-4} \tag{4.1}$$

Entering the above relation directly would result in the creation of many auxiliary variables and many sub-constraints for the multiplications, divisions and linear sub-expressions. Our simplifier is able to infer from Equation 4.1 that $e = 0$. Since this is a very deterministic process, it takes only a fraction of the time the construction and solution of a corresponding constraint expression (by splitting into respective sub-constraints) would require.

Such expressions typically occur in automatically infered and generated constraint models. For enhanced solving performance it is important to find a symbolically simplified equivalent with a potentially drastically reduced number of variables and operations.

After simplification the resulting form can be trivially tested for the required linear property. Linear expressions can be handled directly by the propagator. There is a limited number of propagators that express certain non-linear relations. Other non-linear polynomials not matching any of the provided propagators need additional transformation steps.

In [Apt and Zoeteweij, 2006] an extensive analysis on the handling of arithmetic constraints on integer intervals is given. The authors discuss several approaches of non-linear arithmetic management and splitting, prove their correctness, and finally analyze and benchmark the trade-offs of each variant.

Basing on this work we automatically split non-linear polynomials intro sub-relations of the following types:

[6] This example expression came up in a news-group discussion in `comp.lang.lisp`, message-id `1in6dgh.o04hgmch6wsgN%wrf3@stablecross.com`. It has already been simplified by hand for presentation brevity as the original expression is almost two pages long. The fully expanded original expression is displayed in Appendix A.

- $x = a_1 y_1 + \cdots + a_n y_n + b$ (linear expression)
- $x = y \cdot z$
- $x = y^n$

The resulting relations as computed in the splitting process are then added to the constraint store. They are connected through a number of auxiliary variables and thus represent a constraint conjunction that is equivalent to the original non-linear relation.

This required splitting into sub-constraints connected by variables is the very reason why symbolic simplification yields such enormous gain. Without simplification Equation 4.1 of Example 4.2 would require at least 31 constraints and 62 additional auxiliary variables for connecting them, whereas the simplified term 0 can be trivially expressed.

4.2.3. Global Constraints

Global constraints that operate on multiple variables and prune a number of domains simultaneously are important for a modern constraint solver and for an efficient approach on complex constraint problems, i.e. global constraints implement propagators for *hyper-arc consistency* (cf. Definition 2.14). For CLFD we have implemented a variety of complex constraint propagators to illustrate the simplified addition of such constraints into the framework and to analyze and demonstrate its competitiveness in conjunction with sophisticated pruning algorithms.

The `all-different` Constraint

This constraint is a global constraint that operates on n variables. The propagator has the following signature for its instantiation.

$$\texttt{all-different} \;::\; \textit{var-seq} : \texttt{seq}\,\texttt{var}_{\texttt{int}} \rightarrow \textit{displace-seq} : \texttt{seq}\,\texttt{int} \rightarrow \texttt{propagator}$$

The constraint restricts all variables in *var-seq* to be of mutually distinct values. An optional parameter *displace-seq* is required. This parameter is a sequence of integers that are interpreted as "shift values" for the variable at the corresponding index position. The constraint function thus requires $|\textit{var-seq}| = |\textit{display-seq}|$ as necessary pre-condition. More formally the constraint can be expressed as

$$\texttt{all-different}\ \langle x_1, \ldots, x_n \rangle\ \langle v_1, \ldots, v_n \rangle \Leftrightarrow \bigwedge_{1 \leq i < j \leq n} x_i + v_i \neq x_j + v_j$$

Example 4.3 (A simple `all-different` example)**.** Consider the following instantiation call

$$\texttt{all-different}\ [x, y, z]\ [0, -1, 2] \Rightarrow p$$

This results in the creation of a single propagator p which implies the following constraints

$$\begin{aligned} x + 0 &\neq y - 1 \\ y - 1 &\neq z + 2 \\ x + 0 &\neq z + 2 \end{aligned}$$

All three constraints are enforced simultaneously for all three variables together. This achieves better pruning than a substitution with the corresponding binary constraints.

If *displace-seq* is empty it is interpreted as a sequence with the necessary number of zeros. Since this parameter is optional it defaults to the empty sequence.

Instantiation-based *all-different* This propagator provides a simple yet quite effective algorithm for the all-different constraint. Whenever a variable is reduced to a singleton domain this value is removed from all other participating variables. The propagator thus becomes mostly effective when other propagators have already reduced the domains. Additionally, internal caching has been implemented to avoid unnecessary repeated checking.

Bound-consistent *all-different* This variant of the all-different propagator only reduces variables bounds. This limits its effectiveness but on the other hand requires only a limited computation complexity. The algorithm was first presented in [Lopez-Ortiz *et al.*, 2003].

It is based on Hall intervals of the variable domains.

Definition 4.3 (Hall-Intervall)**.** Consider a CSP $C = \textit{all-different}\ \langle x_1, x_2, ..., x_n\rangle \wedge \bigwedge_{i\in\{1,...,n\}} x_i \in D_i$. The D_i are represented as integer intervals. An integer interval $I = [a, b]$ is a *Hall interval* for C, iff:

$$|I| = |\{X_i : D_i \subseteq I \text{ and } 1 \leq i \leq n\}| \quad \text{or resp.}$$

$$b - a + 1 = |\{X_i : a \leq min(D_i), max(D_i) \leq b \text{ and } 1 \leq i \leq n\}|.$$

The computed Hall intervals can then be used to reduce the domain bounds. Consider a CSP $\texttt{all-different}(\langle x_1, x_2, ..., x_n\rangle) \wedge \bigwedge_{i\in\{1,...,n\}} x_i \in D_i$ and a Hall

interval $I = [a, b]$. The domain D_j of a variable x_j (not part of the Hall interval) with $a \leq min(D_j) \leq b < max(D_j)$ can then be reduced to $D'_j = D_j \cap [b+1, max(D_j)]$. For a second case $min(D_j) < a \leq max(D_j) \leq b$ the lower bound can be reduced accordingly.

Hall intervals are quite simple to compute and can be re-computed cheaply.

Domain-consistent *all-different* The propagator for the strongest variant of the *all-different* constraint is based on [Régin, 1994].

First, a special bipartite graph, a *value graph* is constructed from the input parameters relating variables and values.

Definition 4.4 (bipartite graph)**.** A *bipartite graph* is an undirected graph $G = (V, E)$ where the node set V is divided into two sets V_1 and V_2, such that for all $(u, v) \in E$ implies $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$, i.e. all edges connect only nodes between the two sets V_1 and V_2. We denote a bipartite graph as $G = (V_1, V_2, E)$.

Definition 4.5 (value graph)**.** A *value graph* of an *all-different* constraint C with `all-different`(X_c) is a bipartite graph $G_w = (X_c, D(X_c), E)$ with $(x_i, a) \in E$ iff $a \in D(x_i)$.

By applying the method in [Alt *et al.*, 1991] we find an initial maximal matching covering the variable node set X_c. The algorithm then proceeds to remove any edges that are not part of any maximal matching.

Definition 4.6 (matching)**.** In a bipartite graph $G = (V_1, V_2, E)$ we call a subset of E a *matching*, if no two edges of this subset share the same node.

A matching of biggest cardinality is called a *maximal matching*.

A matching M *covers* a node set X, if $\forall x \in X$ then x is adjacent to an edge.

Régin proved that all edges belonging to a maximal matching must either be part of an alternating path (i.e. alternately connect two nodes of each node set of the value graph) of even length or of an alternating cycle. All other edges connect domain values that are not part of any solution and can thus be removed by pruning the respective domain values.

We explore alternating paths via a simple breadth first search starting with a free, i.e. unmatched, node. Alternating cycles are discovered by exploring the strongly connected components (SCCs) of the value graph. We use a variant of the SCC exploring algorithm [Nuutila and Soisalon-Soininen, 1994] that requires less memory and visits fewer nodes than the standard algorithm.

This last propagator proved to be very efficient and fast, even though it employs complex algorithms.

The Global Cardinality Constraint

The global cardinality constraint (GCC) is ubiquitous in situations where a selection of values has to be constraint to occur within a certain number of times. Such problems include the creation of special value sequences or in rostering problems. GCC creation is described by the following signature.

$$\texttt{gcc} \;::\; \textit{var-seq} : \texttt{seq}\,\texttt{var}_{\texttt{int}} \rightarrow \textit{values} : \texttt{seq}\,(\texttt{int}, \texttt{int}, \texttt{int}) \rightarrow \texttt{propagator}$$
$$\texttt{gcc} \;::\; \textit{var-seq} : \texttt{seq}\,\texttt{var}_{\texttt{int}} \rightarrow \textit{values} : \texttt{seq}\,(\texttt{int}, \texttt{var}_{\texttt{int}}) \rightarrow \texttt{propagator}$$

The variables of *var-seq* are constrained with respect to the *values* sequence of integer triples. Each triple is interpreted as $(value, min, max)$ in such a way that all variables together must be assigned to *value* at least *min* times and at most *max* times.

A second variant of the GCC exists, where the number of occurrences of a *value* is restricted depending on an integer variable. Initially, the lower and upper bound of the variable's integer domain is taken into account which is then incrementally reduced to a singleton domain during the solving process.

In [Quimper *et al.*, 2004] an algorithm for GCC pruning is presented. We adapted this work for our propagator. The algorithm essentially uses the propagator of the *all-different* constraint as a black box and decomposes the GCC constraint into a number of interdependent *all-different* constraints. However, this method would result in a lot of value copying and repeated graph construction. The algorithm is therefore reduced to a more efficient representation as a flow network (cf. [Ahuja *et al.*, 1993]). This network flow problem is then used for incrementally finding maximum matchings to remove unmatched domain values.

4.2.4. Predicate Function Constraints

An important feature for the transparent integration with the standard Common Lisp system is the constraint coercion of ordinary Lisp predicate functions into constraints.

A predicate function (i.e. an ordinary Common Lisp function returning only a Boolean value) is called as a higher order function (more specifically a closure) that determines whether a set of conditions is satisfied.

Since function application requires the instantiation of all participating variables to a single value the predicate propagator employs a technique very similar to

residuation [Aït-Kaci and Nasr, 1989]: The propagator is suspended until all but one variable are bound to a valuation domain. Now the predicate function can be applied and tested to every value of the remaining multi-valued domain. This domain is then reduced to values that did not fail the predicate test.

The advantage is that every COMMON LISP function returning a Boolean value can be easily transformed into a constraint which enables a very natural Lisp-like experience. Such constraints cannot be used to reduce the search space since they are not able to reduce values of constraints directly by calling a domain reduction function, but they can function as an assertion that excludes certain conditions by signaling a failed state. Furthermore such a function may add additional constraints to the constraint system and thus may overcome the limitation of missing direct variable interaction.

4.3. Reified Constraints

Reified constraints simplify the modeling of many constraint problems, specifically since they can act as some kind of conditional. The general definition is as follows

$$C \Leftrightarrow var_{bool}$$

The relation between the constraint C and the Boolean variable can be expressed semantically with a simple four-step pattern. Note that the Boolean variable is expressed as bit values and thus a sub-type of an integer variable to enable the easy integration with other numeric constraints as is typically required.

1. if C is entailed $\Rightarrow var_{bool} \in \{1\}$,
2. if $\neg C$ is entailed $\Rightarrow var_{bool} \in \{0\}$,
3. if $var_{bool} \in \{1\} \Rightarrow$ add C to store,
4. if $var_{bool} \in \{0\} \Rightarrow$ add $\neg C$ to store.

CLFD supports reified constraints in three forms. Steps 3. and 4. are obviously easy to implement so the main difficulty is with the entailment of C.

A first approach is a specific propagator that algorithmically describes the complete $\Leftrightarrow$ relation for a particular constraint C. This is not different from any other constraint propagator. It (potentially) provides the most efficient pruning approach by tailoring its checks directly to a pre-known C but it is also the most complex and time consuming approach for an implementer.

The second approach requires an entailment-test method that must be provided by the propagator for C that must be able to check whether C or $\neg C$ holds. Ideally, the provided test method will be able to infer entailment before all variables of C are bound to speedup the solving process.

The third approach is the most general, but also the least efficient and thus slowest method. However, it can reify every constraint as long as the user is able to provide a negation of C[7]. The idea is directly related to the integration of standard function by residuation as described in the previous Section 4.2.4.

The entailment test of C is delayed until all variables $var(C)$ are bound to the singleton domain. The test is then only a matter of creating a new solver (sub-)configuration that only consists of $var(C)$ and C itself (or $\neg C$ for the second case) and propagate it to initiate the constraint test. The variable var_{bool} can then be set accordingly in the original propagation, together with the insertion of C or its complement.

4.4. Change Functions

Propagators for complex constraints such as all-different or global cardinality need to maintain intricate data structures. For example the bipartite value graph of GCC and all-different must be kept in synchronization with the domain values whenever the propagator is executed. This means that the propagator either constructs a new value graph for each propagator run or iterates through all domains and deletes graph edges where other propagators have already pruned the corresponding values. This is a time consuming process and one would want a way to incrementally make adjustments to the graph whenever a value is removed from the domain by a propagator.

The *change functions* are a way to realize such incremental adjustments. They are an extension of the *notifier* concept as present in SCREAMER which is there used solely for propagator scheduling.

Technically, they are simply closures that are created by the propagator at instantiation time and are then associated with a single variable and a domain modification event (cf. Section 3.2.2). In reaction to domain reduction they are called immediately.

The signature of a change function cf is quite simple.

$$cf \ :: \ \texttt{propagator} \rightarrow \texttt{domain} \rightarrow arg \rightarrow \texttt{propagator}$$

[7]For a number of standard constraints their negated relation can be infered automatically by the solver, similar to the standard `complement` of COMMON LISP.

It is called with the current state of the propagator instance that initially created cf. Additionally, the variable domain and the argument *arg* which will be part of the call to the domain reduction function (this is either a number or a domain, depending on the reduction function) are supplied. The result of a call to cf is a propagator with an altered internal state.

For an example application consider the domain-consistent all-different constraint. The internal value graph consists of edges (v, d) between variable nodes V and value nodes D iff the associated value of a value node $d \in D$ is a domain value of $v \in V$. The all-different propagator may install a change function for every node in V that reacts upon value removal and deletes to respective edge of the value graph. The value graph would be in a consistent state with respect to the domain values when the propagator is executed.

This would eliminate the need to iterate through all domain values of every variable to remove the edges to value nodes of non-existing domain values. The alternative, to create a new value graph for every propagator execution is even more costly.

Since a change function is specialized to a particular variable and a particular propagator instance, it can close over very specialized information such as specific index positions in an internal vector as variable bindings are captured within the created closure. This allows for the generation of very efficient code for the change functions and ensures their fast execution.

It is important that change functions may only alter the internal data structures of a propagator. They must not initiate any domain pruning events or call other propagators to ensure overall correctness and termination.

4.5. Further Work

Further work in connection with a supervised student's thesis [Niesche, 2007] investigated the implementation of finite set constraints by evaluating several domain representations and the definition of a number of accompanying propagators. Basing on [Azevedo, 2002] a second aspect was the investigation of propagator optimization by dynamic generation compilation of problem-specific propagators.

An implementation of a simple finite set representation has been realized. Additionally a number of propagators for the basic set operations are provided, though were only lightly tested and no steps have been taken to analyze and enhance their performance in any way.

The result of a second student's work [Cerwinski, 2007] was two-fold. First, an implementation of the *regular language constraint* [Pesant, 2004] was sought

by making use of the finite integer domain the propagation architecture of the framework. This work has been realized and is the foundation for a sub-sequent extension.

The global constraint catalogue [Beldiceanu *et al.*, 2005] is an extensive collection of many global constraints. The algorithmic behavior of several global constraints is described in terms of automata. A second aspect of investigation was to derive a textual representation of these automata and translation of each such automaton to an instance of the regular constraint. This would make it possible to derive propagators for a large number of global constraints without investing the time to implement dedicated propagators for each such global constraint, but of course at the trade-off of less efficiency. This second part has seen some initial work but only limited testing. Overall, the approach has been found to be feasible and represents a way to quickly realize complex global constraint algorithms for pruning even though the result implies some performance penalties.

5. Search

As has been illustrated in the previous chapter, the constraint propagation phase can only achieve arc-consistency (and hyperarc-consistency) or bounds-consistency, depending on the deployed and available propagators. However, propagation alone is not complete and thus not sufficient to generally infer solutions.

In this chapter we will introduce a generic search framework in Section 5.1. Afterwards, Section 5.2 gives a detailed analysis and specification of the individual components. Later in Section 5.3, we describe a number of search algorithms and goals that were realized within our framework by making use of the composition and inheritance capabilities of the individual search aspects. Finally, Section 5.4 is concerned with implementation and abstraction details concerning variable synchronization and Section 5.5 details on problems that arise due to the possibility of the CLFD framework to integrate different (and incompatible) domain types.

The Necessity for Search Consider the Send-More-Money problem as depicted in Figure 5.1. The eight variables are initially assigned to the set of integers $\{0, \ldots, 9\}$. They are constrained to be distinct and to fulfill the arithmetic expression which connects the individual variables. The figure also shows the unique solution for this CSP. Unfortunately, simple propagation of the arithmetic and distinction constraints is not sufficient to arrive at the desired solution.

Figure 5.2 shows the detailed steps of the solution process. After an initial constraint propagation phase only three out of eight variables are bound to singleton values. At the same time, no additional information can be inferred from the

$$S, E, N, D, M, O, R, Y \in \{0, \ldots, 9\} \quad \wedge \quad S, M \neq 0 \quad \wedge$$
$$\textit{all-different}(\{S, E, N, D, M, O, R, Y\}) \quad \wedge$$

$$\begin{array}{ccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array} \quad \Rightarrow \quad \begin{array}{ccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

Figure 5.1.: SEND-MORE-MONEY Problem and Solution

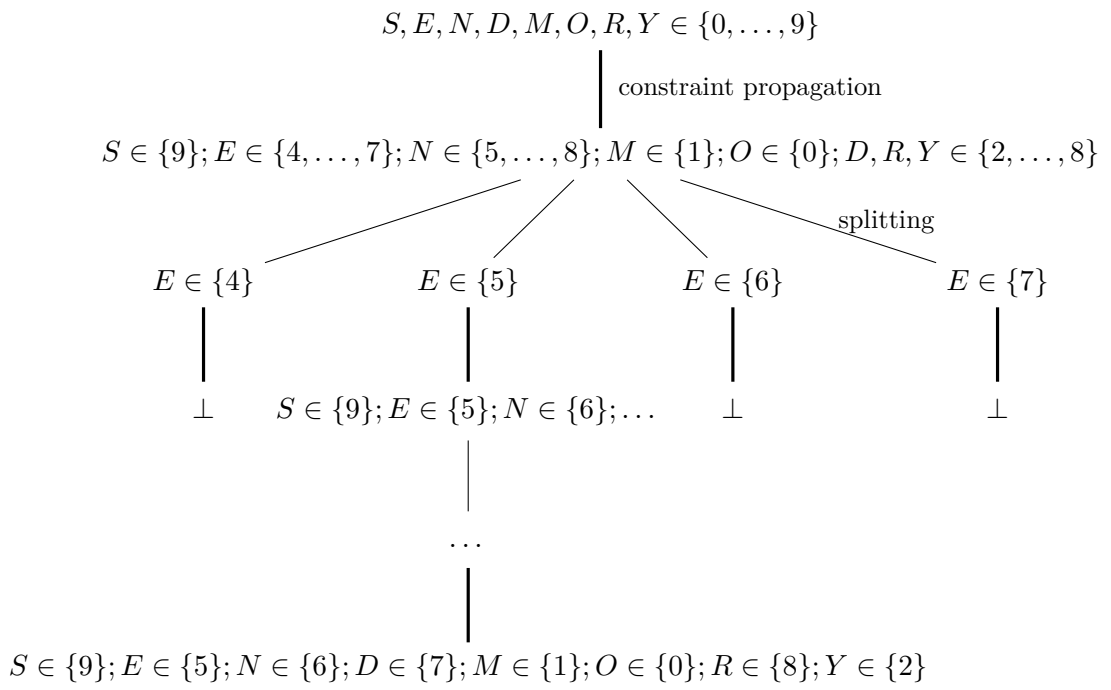


Figure 5.2.: Example search tree for the SEND-MORE-MONEY problem.

constraints. This leaves us with the only possibility of try-and-error to eventually arrive at a solution. Here, we randomly selected the variable E and test for every single element in its domain whether their assignment leads to a consistent CSP.

The splitting into individual domain values leads to a branching of possible assignments. With this new information due to the reduced domain of E, it is possible to further reduce domains by constraint propagation (or to find the resulting CSP inconsistent). The process continues until all variables are assigned to a valuation domain. The resulting tree structure is called the propagation-and-search tree of a CSP (or search tree for brevity).

In our example only one path of the tree leads to a solution, all other leaves, i.e. all other assignments, resulted in an inconsistent CSP and are thus fail nodes in the tree.

Formally, we are splitting a constraint problem into two equivalent sub-problems. According to Definition 2.11 this is allowed and the union of the solutions is the solution set of our original CSP. Thus we do not lose any solutions with respect to the original CSP. In correspondence to [Zoeteweij, 2005] we denote the function that splits the domains as *domain branching function*.

Definition 5.1 (domain branching function)**.** For a CSP wrt. Definition 2.8 a

domain branching function is a partial function

$$dbf : t_1 \times \ldots \times t_n \to \mathcal{P}(t_1 \times \ldots \times t_n)$$

such that for $\langle D_1, \ldots, D_n \rangle \in t_1 \times \ldots \times t_n$ where t_i is a domain type and with

1. $\forall D_i \neq \emptyset$, and
2. $\exists D_j$ which is not a singleton domain

then $\{D'_1 \times \ldots \times D'_n | \langle D'_1, \ldots, D'_n \rangle \in dbf(D_1, \ldots, D_n)\}$ is a proper and minimal covering of $D_1 \times \ldots \times D_n$. A *covering* of a set X is a set of subsets of X whose union equals X. A *proper* covering does not contain X itself.

5.1. A Generic Search Framework

We have seen that search is a necessary step in our solver framework. However, a simple search procedure as described in the above example is usually not recommended because it will either result in poor performance or may even not be able to deliver the sought result because of time and memory limitations. The design goal of a search method is the fast computation of a solution by using only limited resources. Ultimately, the search tree is sought to be as small as possible to narrow the required memory. Generally, this is accomplished by assessing the success probability of a certain path within a search tree with the help of heuristic quantification.

Many search methods and heuristics have been developed for specific problem domains. Since they typically are derived from one another and thus share properties and implementation details, this relationship should also be expressed in our implementation. Not only does this ease implementation and theoretical analysis, it is also good engineering practice for long-term maintainability.

The work in [Hölzl, 2001] develops a model for integrating constraint solving and lambda calculus. A realization of this model also describes a protocol for search algorithm derivation [Hölzl, 2002]. However, it turns out that the presented protocol design severely limits the capabilities of a search algorithm and does not cater to the properties of more involved algorithms other than simple depth-first search or breadth-first search. We therefore developed our own search framework [Frank, 2007] that is the base for the expanded variant as described in the following sections. Our overall design is influenced by our strategy approach of META-S as represented in [Frank and Mai, 2002; Frank *et al.*, 2003a; Frank *et al.*, 2005]. The new design has been verified to be highly adaptable by describing a variety of search methods within CLFD.

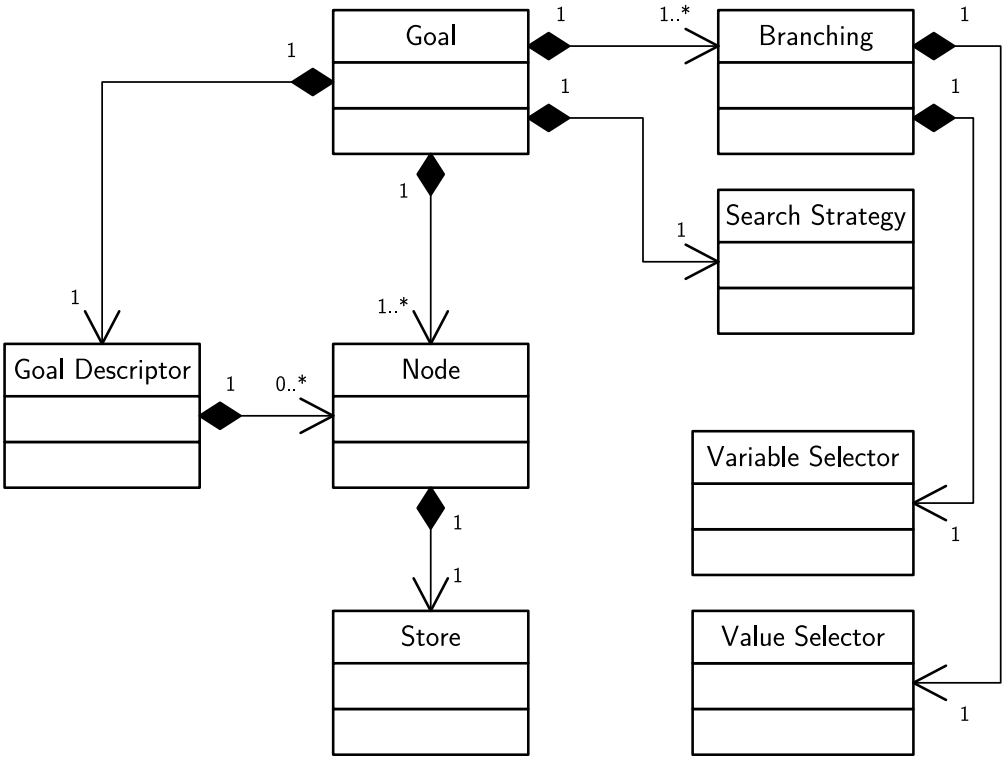


Figure 5.3.: Generic model of the search framework.

The general elements of our search component are depicted in Figure 5.3. We distinguish the following components:

goals A goal describes in terms of a collection of nodes (see below) *what* exactly constitutes a solution. Examples of very simple goal types are single- and all-solution goals, i.e. a collection of 1 or n nodes respectively.

exploration strategy A strategy encodes the order for the exploration of sub- and sibling nodes.

selection heuristics The heuristics (variable and value selector) steer our movements within the search tree such that they govern which sub-tree is about to be computed next.

nodes A node encapsulates a constraint store plus some further heuristic information. This allows for additional data attachments and choice-point management actions than a store alone would be able to provide. A node represents one choice-point configuration within the non-deterministic search process.

We will detail the individual search components in the next section before we describe their algorithmic interaction.

5.2. Search Components

The components of the search framework allow the comfortable realization of many search algorithms by basing them on an already implemented algorithmic method. Often only one or two of the protocol methods have to be adapted to realize a different search algorithm or search goal.

5.2.1. Know Your Goals

A goal ultimately describes what is considered a completed solution for a given CSP. Recall that a solution until now simply was described as a CSP where all variables are assigned to a singleton domain, thereby forming a consistent CSP with respect to the other constraints of the constraint system. However, this is simply an assignment that fulfills our notion of a solution, but our *goal* may be more pronounced, e.g. finding the first, last, all or best (wrt. to some objective function) solution(s).

A goal is described by the following protocol interface as expressed by generic functions. First, we need a way to initialize a goal, e.g. by assigning a root node or setting up internal management data structures.

initialize-goal (*goal strategy node*) [*Generic Function*]
(*:method-combination and*)

The initialization method fills internal data structures and uses node as root of the search tree. Initialization typically also consists of a first propagation cycle and additional constraint insertion. This cycle can possibly fail, and it is therefore required to return t upon a successful return. The methods of an inheritance lattice are all evaluated and combined by the and method combination.

A goal step consists of the initiation of the constraint propagation phase. At this point other actions can be triggered, such as additional constraint insertion or data display for debugging aid.

goal-step (*goal node*) [*Generic Function*]
(*:method-combination and*)

Initiates the constraint propagation phase on node. It may also add additional information or constraints before propagation.

The next method decides whether a goal has reached its intended final step:

goal-reached-p (*goal*) [*Generic Function*]

A predicate method that returns t if a goal's solution is reached.

Some goals may require a re-start from the tree root. This can be realized by an adoption of the following method.

goal-restart (*goal strategy*) [*Generic Function*]

This method initiates a restart of a goal from its initial search root.

Additionally there are two methods collect-solution-node and process-fail-node which can be advised to handle solution or fail nodes. This may be used for statistics collection or display of the computation tree. Note, that the functional description of the protocol methods is quite abstract. This is caused by the high level of abstraction provided by this interface which is able to express a wide range of search goals and search algorithms. Figure 5.5 displays all currently available goals which were implemented through the above protocol. Further details on the individual goal protocol steps are detailed in [Frank, 2007] and Section 5.3.

Algorithm 5.1: The general algorithm for the goal based search and propagation process.

Data: $G = (\mathcal{WS},\mathcal{S})$ is a goal
Data: $\mathcal{WS} = \emptyset$ is a work set of nodes
Data: $\mathcal{S} = \emptyset$ is a goal set of nodes
Data: n is the inital node, the root of the search tree

```
1  WS ← {n}
2  while ¬(reached G) ∨ restart G do
3  |  foreach s ∈ WS do
4  |  |  propagate on node s
5  |  |  if s is a solved node then
6  |  |  |  S ← s ∪ S
7  |  |  else
8  |  |  |  split s and add the resulting s_i to WS
9  |  |  end
10 |  end
11 end
12 return S
```

The Algorithm 5.1 describes the general search approach based on our goal abstraction. Remember that a node can be portrayed as a store associated with heuristic variable and value selector functions. Furthermore, a goal can be abstracted to a two-tuple of two sets. A work set $\mathcal{WS}$ is initially populated with a single node which describes the root of the search tree. Secondly, the set $\mathcal{S}$ is our result set of solution nodes. The kind of a goal, e.g. single solution or best solution, controls the number and kind of nodes considered to be part of set $\mathcal{S}$. The algorithm runs until there are no more nodes in the working set to be considered for constraint propagation. If a propagation results in a non-solution node (i.e. not every variable is bound to a singleton domain), it is split into sub-problem nodes that are added to the work set. A goal may also be re-started with a modified work set.

Now we want to customize this search algorithm in several ways through:

- a predicate describing whether a goal is reached or not,
- a predicate that decides which solution nodes are part of the result set,
- the order in which the nodes of the work set are explored one by one

```
1  (defun search-goal (goal strategy initial-node)
2    (if (initialize-goal goal strategy initial-node)
3        (let ((strategy (goal-strategy goal)))
4          (loop while (or (goal-restart goal strategy)
5                          (nodes-left-p strategy))
6                do (let ((node (next-node strategy)))
7                     (if (goal-step goal node)
8                         (let ((var (select-next-branching-var node)))
9                           (if var
10                              (choose-on-nodes goal
11                                               strategy
12                                               (funcall (branch-value-selection
13                                                          (current-branching node))
14                                                        var
15                                                        node))
16                              (collect-solution-node goal node)))
17                        (process-fail-node goal node)))
18               until (goal-reached-p goal)))
19        (process-fail-node goal initial-node))
20   (return-goal-result goal))
```

Figure 5.4.: The generic search function that connects all components of the search framework.

and several other aspects. This can be achieved by implementing or changing at least some of the above protocol methods.

All the protocol methods as introduced above are combined to a generic search function `search-goal` which realizes the complete search phase. It makes use of other components of the search framework, but this usage is mostly hidden so we will illustrate these functions already at this time. The general algorithm is the one already defined in Algorithm 5.1. The difference is that through the combination of the individual goal protocol methods we are now able to adapt the various properties of a goal and can describe new types of goals with different properties.

Figure 5.4 shows the code for the generic search function. After the goal initialization in Line 2 the search phase in fully contained in Lines 4–18. The search phase is started if there is at least one node that can form the root of a new search sub-tree (Line 5) or the goal may be restarted (Line 4). After retrieving the node the propagation phase is initiated (Line 7). This propagation may fail, so we process the resulting fail node (Line 17) or continue with the resulting consistent node. This result node is now either already a solution, which is then handled in Line 16, or further search is necessary. In the latter case (Lines 8–15) then generate the corresponding child nodes of the search tree.

The search phase is finished when the goal is reached (Line 18) or when there are no more nodes left to process (see Line 5).

We will argue in Section 5.3 that this combination of a goal protocol and a generic search function is able to describe a wide range of different search methods, that can be realized by exchanging a subset of the protocol methods of an already implemented algorithm.

5.2.2. Traversal Abstractions

During the search phase we decide to follow a path in the search tree for a potential solution. However, it is not clear whether the final leaf node of the tree will result in an in-consistent CSP or in a solved CSP where all variables are assigned. Therefore, it might be necessary to backtrack and explore a different path. For this operation, a solver must be able to retain a previous state for an exploration re-start.

In [Choi *et al.*, 2001] a comprehensive overview of tree exploration components is given. The authors describe the aspects: state restoration, a branching descriptor, and an exploration strategy. These components are also reflected in our framework (cf. Figure 5.3), though the need to deal with potentially different domain types in a single CSP, as in our framework, requires the consideration of further aspects and makes the model more intricate. This will be worked out in detail in the following.

State Restoration We need a state restoration policy to go back to a previous store state during the search tree exploration. This property is reflected by two components in our framework: a node and its accompanying store. A node must be able to restitute the store it subscribed to. It may rely on the store's own capability to retain state or retrieve the corresponding store state itself.

For example, when applying a restoration policy that fully copies states upon modification, a newly derived store is a full copy of its previous state. In the case of a trail-based approach [Schulte, 1999] where the applied changes are recorded, it is the responsibility of the node to replay the respective changes starting at a back-up node.

We have realized trail-based, full copying and lazy copying policies. The last policy reduces memory consumption by a copy-on-write approach. The copying approaches have shown to provide the best performance.

Exploration Policy This component simply decides which node should be explored next. There is an obvious choice between child, sibling or parent, but one may also iterate randomly on already discovered yet unexplored nodes. The *search strategy* (or traversal strategy) simply provides a data structure for ordering and

retrieving the created tree nodes. Remember, that a new node is created when the propagation of the current node does not yield a solved or failed CSP. This is realized by Lines 8–15 in Figure 5.4.

The exploration policy results in the visiting pattern of the search tree in depth first, breadth first, random, or similar order. Consider for example two simple variants of internal management data structures:

Stack If the nodes are managed in a stack then the last nodes added will be retrieved first. This results in depth first search ordered traversal of the tree.

Queue If the nodes are managed in a queue, then new nodes are added at the end of the queue. Earlier nodes are retrieved first from the queue. This means that nodes a traversed in breadth first order resulting in a breadth first search of the tree.

Branching Descriptors A *branching* describes the potential children of a tree node [Choi *et al.*, 2001]. It encapsulates the variable and value selection functions described in the next section. Furthermore it holds the variables these selection functions apply to. In the simplest case thus all un-instantiated variables of the CSP belong to a single branching descriptor which also assigns a single variable and value selection function. A branching is said to describe the children of a node because the application of the selection heuristics as assigned to a branching will result in the creation of new child nodes and thus new sub-trees.

All initial user-defined variables must belong to a unique branching. Since the selection functions are specific to the domain type (especially the value selection function) a CSP may consists of more than one branching descriptor (at least one for each variable type) which are then processed in goal-dependent order.

5.2.3. Selection Heuristics

The selection functions ultimately describe the overall gestalt of the search tree. A branching first applies the *variable selection* function to its set of un-instantiated variables. In a second step the *value selection* function is applied to the chosen variable v. The result is a set of nodes where each node v is constrained to a reduced domain value set compared to its original one. To summarize, the selection functions have the following signature:

$$\begin{aligned} \texttt{variable-selection} &:: [\texttt{var}] \rightarrow \texttt{var} \\ \texttt{value-selection} &:: \texttt{var} \rightarrow [\texttt{node}] \end{aligned}$$

Since the selection heuristics are not very complex and thus easy to implemented and exchange, we provide a number of selection functions for integer variables.

Variable selection:

- **listed:** select in the order as listed in the branching descriptor.
- **min/max size:** select the variable with the smallest/largest domain size.
- **most constrained:** select the variable with the most propagators attached to it.
- **largest/smallest bound:** a number of selector functions that chose the variable with the smallest/largest lower/upper bound.
- **random:** select a random variable.

Value selection:

- **split:** creates two child nodes each containing the lower and upper half of the variable domain, respectively (for integer domains and floating point interval domains).
- **min/max:** creates two child nodes, one containing a variable with a singleton min or max domain value respectively, and a second node where the variable contains the remaining domain values.
- **label:** creates n child nodes, one for each domain value (n is the domain size).
- **random:** creates two child nodes, one containing the selected variable v with a singleton domain of a random domain value, the second node contains v without the domain value from the first node.

Note that the *min/max* and *label* selection functions are quite similar. The label method results in a search tree where a node has many direct children in one tree level, whereas the *min/max* function creates an almost equivalent tree but it distributes the nodes to a number of tree levels. In fact, [Hwang and Mitchell, 2005] show that for a large number of example problems none of the two methods has any advantage over the other approach.

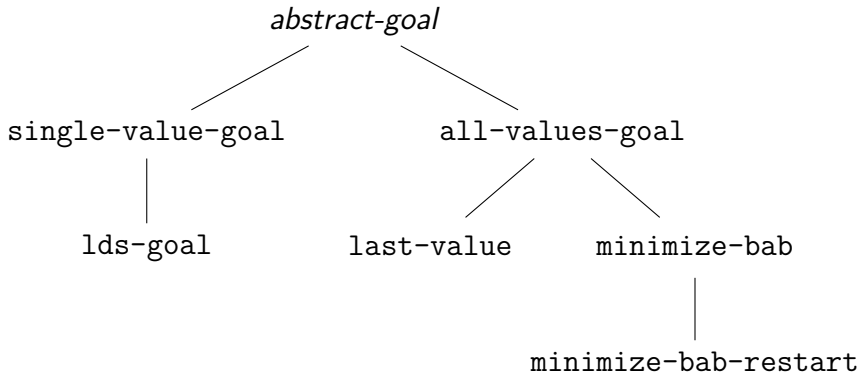


Figure 5.5.: The hierarchy of search algorithms (i.e. search goals) as currently provided by CLFD.

5.3. Search Algorithms

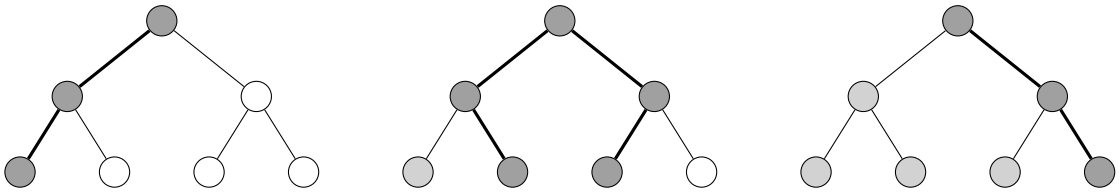
Using the components from the previous sections we can describe a number of search algorithms within our search framework. Figure 5.5 displays the hierarchy of defined search algorithms. The inheritance hierarchy in this case is based on the class hierarchy of *goals* from which the individual search goals are derived.

The most trivial goals are `single-value-goal` and `all-values-goal`. The latter one simply traverses the complete search tree and collects all solved CSP nodes. The first goal returns the first solution node that is discovered during tree exploration. Remember, that the order in which nodes are created and visited depends on the selection heuristics which are not part of a general goal description but are assigned during the creation of an initial CSP configuration. The `last-value` goal is a variant of the `all-values` goal that only returns the single goal that has been found last when exploring the search space.

Branch and Bound The `bab` goals implement a branch-and-bound scheme [Land and Doig, 1960] which finds the best solution with respect to a user-defined cost function. This cost function is expressed as a constraint whose cost value is defined on a special cost variable. For each goal-step a constraint that describes the currently found best cost value is added to the CSP node.

A second `bab` goal provides a modification which restarts the search from the root after a solution has been found. The advantage of this method is, that the cost variable does not need an additional constraint at each propagation step but is assigned only once in goal-restart since this additional constraint is then already required for the whole search tree.

It may be surprising that the branch-and-bound goal is derived from a goal



(a) Round 1, $k = 0$ (b) Round 2, $k = 1$ (c) Round 3, $k = 2$

Figure 5.6.: Search trees for the successive rounds of LDS.

that collects all solutions, since it will only return a *single* best solution. However, this goal has to traverse the complete search tree (even though this tree has been reduced by several sub-trees due to the additional constraints) to prove that the current solution is the best one. Therefore, the already provided protocol methods from the `all-values` goal which explore the complete search space minus the branches that are cut off by the cost function, provide a matching foundation.

Limited Discrepancy Search The limited discrepancy search scheme (LDS) [Harvey and Ginsberg, 1995] has been shown advantageous for scheduling problems and similar settings. The fundamental concept is the assumption that the selection heuristics is more probable to be wrong when applied to nodes closer to the root, since variable domains are less pruned at this stage. However, it is assumed that these heuristic selections are correct in general. We therefore allow the node selector of the exploration policy to decide *against* its usual choice for k times. This creates a number of paths that do not cover the complete search tree. If no solution is computed, the goal is restarted with an increased discrepancy level k.

Figure 5.6 illustrates the possible paths for three search rounds and three discrepancy values. This algorithm is bound to re-compute nodes that have already been derived at a previous stage. However, despite this additional computation work this method has been successfully applied. Sophisticated implementations of this search scheme sometimes provide a cache of already computed nodes to reduce the re-computation work.

5.4. Variable Synchronization

We have already described in Section 3.2 that variables reside in the store. It is responsible for channeling any operation that may affect a variable or its domain. In Section 5.2.2 we introduced a state restoration policy that manages the back-up of nodes and their assigned stores in case of a possible backtracking step. Adhering

to our modular design the overall protocol does not make any assumptions about how this state restoration is accomplished.

However, due to this abstract view we are now facing a performance problem. Variables and their domains are part of the store state that requires backup. Variables are also referenced from multiple components, especially from the store and from propagators. To access a variable from a propagator, the right solution for our abstract storage scheme would be to channel every access through the store which is then responsible for retrieving the correct variable state. This additional indirection comes with a major performance penalty, especially in cases where a variable is simply read, but not modified. We therefore would like to reference variable instances directly from a propagator. Unfortunately, this represents a synchronization problem since the store (and the node abstraction in the upper layer) is responsible for state backup and retrieval.

The problem is especially grave, since a propagator does not (and should not) know anything about the current state restoration policy. We therefore need a mechanism that ensures that the variable instances referenced by a propagator are the correct ones wrt. to the current store.

As an example, consider a full copying restoration scheme, i.e. when a new node is created, the store, its variables, domains and (stateful) propagators are fully cloned. Since variables are referenced by both the store and the propagators, one would possibly apply a method similar to the handling of cyclic references in the Cheney copying garbage collector [Cheney, 1970; Jones, 1999]. The individual steps are also illustrated in Figure 5.7; the initial situation is displayed in Figure 5.7a:

1. All variables as referenced by the store are copied. Each variable saves a forward reference to its copy (Figure 5.7b).

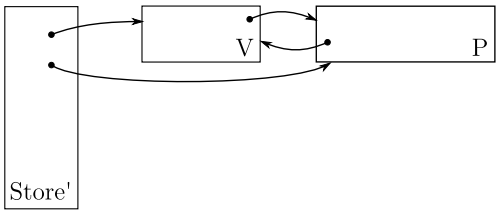
2. Now, all propagators are copied step by step (Figure 5.7c).

3. All variables referenced by a propagator copy are still the old variables. The forwarding references are used to update the references in the propagator copy to the new variable copies.

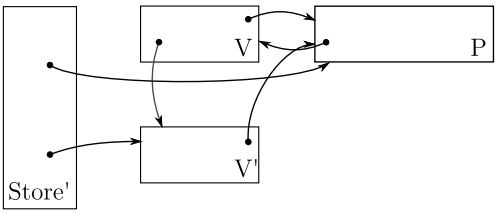
4. At the same time the propagator reference within the variable copy is updated to the new propagator copy (Figure 5.7d).

However, this scheme only works because we assume knowledge about a fully copying backup policy at the level of variable and propagator data types. Our protocol does not allow such assumptions because storage retrieval is a property of the store, not a property of a propagator or variable.

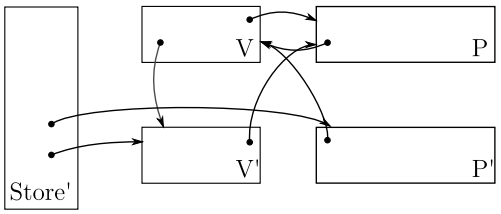
To allow direct access of variable instances from a propagator, an additional synchronization policy had to be established that ensures that a propagator only



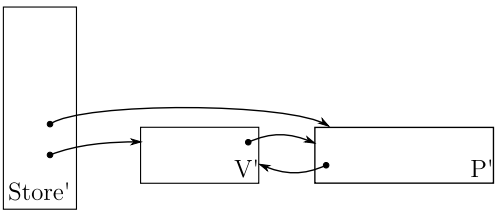
(a) The initial situation were only the store is already cloned.



(b) The variable copy and forward reference is created.



(c) The propagator copy is created.



(d) The variable references in the propagator copy are updated using the forward reference. Also update the propagator as referenced by the variable copy.

Figure 5.7.: The main steps of a Cheney-based copying scheme.

executes when all its references point to correctly set up variable instances. The store runs a synchronize-variables method for the propagator before its execution and also caches the result to avoid further unnecessary synchronization runs.

It is the responsibility of the propagator developer to mark all slots as `:synchronized` that may contain variable references.

For example consider the following (simplified) excerpt from the *all-different* propagator:

```
(defclass all-different-propagator (propagator)
     ((vars :initarg :vars
            :type sequence
            :synchronized)
      ...))
```

The additional keyword `:synchronized` automatically generates a synchronization method that is executed before the initiation of the *all-different* propagator. This additional slot definition keyword and automatic method generation is realized by the meta-class already depicted in Figure 4.2 and the CLOS meta-object protocol.

$=, \neq, <, \leq, >, \geq$	:: `interval` → `interval`
in	:: `interval` → `interval` → `interval`
$+, -, *, /, min, max, **$	:: `interval` → `interval` → `interval`
$-$, *square*, *exp*, *log*, *abs*, *floor*, *ceil*, *sign*	:: `interval` → `interval`
sin, *cos*, *tan*, *asin*, *acos*, *atan*	:: `interval` → `interval`
sin2pi, *cos2pi*, *tan2pi*	:: `interval` → `interval`

Table 5.1.: Relations and operands on interval variables.

5.5. Interval Constraints in $\mathbb{R}$

So far, we have mostly reflected on domains of integers even though the solver framework is designed to support various finite domain types. A missing piece was the handling of different variable types during search. As we have illustrated in Section 5.2.2, the branching operation for choice-point creation requires selection functions which are tailored to the specific domain type of a selected variable. We are now at a stage where we can fully support other domain types and their respective constraints and propagators.

Besides integer constraints, operations on floating-point intervals are widely used. The integration of the interval domain is supported due our heavily modularized design.

The interval components are used for interval arithmetic on real numbers (in particular double-floats). It uses the sound math library[1] (*smath* library) by Timothy J. Hickey *et al.*, an ANSI C library providing the basic narrowing functions for interval arithmetic. Its theoretical foundations are presented by their authors in [Hickey *et al.*, 2001]. They rely on the rounding requirements for floating point numbers as outlined in [IEEE, 1987].

The interval domain provides a variety of relations and operators. The overall signature is presented in Table 5.1. Since the domain is not countable, the selection heuristics need to account for this limitation. The typical splitting strategy on value selection is a simple partitioning of the interval domain into two equally large sub-domains.

Due to the nature of the interval narrowing approach of *smath*, operators such as *sin* have an incoming argument (the operator's argument variable) and a result variable that reflect the computed result of the operation. That mean effectively for each operator appearance such as $\sin(X)$, internally a constraint relation $aux = \sin(X)$ is created, where *aux* is a generated auxiliary variable. Thus complex

[1] `http://interval.sourceforge.net/`, last visited 2010-04-21

relation/operator	preprojected/converted version
$X\ in\ [Y, Z]$	$X \subseteq \text{gen-var} \wedge Y \leq \text{gen-var} \wedge \text{gen-var} \leq Z \wedge Y \leq Z$
$X > Y$	$Y < X$
$X \geq Y$	$Y \leq X$
$Z = log(X)$	$Z = log(X) \wedge 0 < X$
$Z = X - Y$	$X = Y + Z$
$Z = X/Y$	$X = Z * Y \wedge Y \neq 0$
$Z = X\hat{}Y$	$Z = X\hat{}Y \wedge integer(Y)$
$Z = X ** Y$	$Z = exp(Y * log(X))$
$Z = sin(X)$	$Z = sin(X) \wedge Z \subseteq [-1, 1]$
$Z = cos(X)$	$Z = cos(X) \wedge Z \subseteq [-1, 1]$
$Z = asin(X)$	$X = sin(Z) \wedge abs(Z) < \frac{\pi}{2}$
$Z = acos(X)$	$X = cos(Z) \wedge Z \subseteq [0, \frac{\pi}{2}]$
$Z = atan(X)$	$X = tan(Z) \wedge abs(Z) < \frac{\pi}{2}$
$Z = sin2pi(X)$	$Z = sin2pi(X) \wedge Z \subseteq [-1, 1]$
$Z = cos2pi(X)$	$Z = cos2pi(X) \wedge Z \subseteq [-1, 1]$
$Z = sign(X)$	$Z = sign(X) \wedge Z \subseteq [-1, 1]$

Table 5.2.: Transformed constraint forms of external relations/operators

operator expressions are broken up into their basic operation parts and these fundamental structures are then interconnected via their input and output values thus building a complex network of simple relations.

Constraint Conversion Several constraint relations and operators provided by the interval propagators need a special conversion applied to the input to ensure correct boundaries and types of the results or to base their definition on other basic operators (e.g. $>$ is defined based on $<$ and $\hat{}$ needs a special *integer* type constraint on its second argument to ensure correct results). This is due to the inner workings of the interval arithmetic library which cannot infer these additional constraints *per se*. Omitting these additional rules may lead to poorer or even wrong results, for example in only partly adjusted boundaries. Table 5.2 summarizes the applied conversions as derived from and required by [Hickey *et al.*, 2001].

6. Termination

Since a user is able and desired to define his own solver modules and extensions it is desirable to analyze the necessary requirements that lead to a terminating solving process. Furthermore we will elaborate on the conditions for a consistent (in the sense of repeatable) solution derivation in the case of multiple solutions. Similar to the descriptions of the different solver modules in previous chapters we will consider termination properties in a bottom-up way.

Overall, the whole solving process can be viewed as a fix-point iterating solution finding where several nested inner fix-points are computed in-between – the final result in our case is either a completely instantiated domain set (i.e. every variable is bound to a singular value) or at least one domain became empty, i.e. there is no solution.

We will first introduce the necessary notions and definitions. After that we will elaborate on the application to our problem domain.

6.1. Fix-point Computation

Most of the notions and definitions introduced in this section are primarily derived from [Hein, 2002] and [Pepper and Hofstedt, 2006; Cai and Paige, 1989]. In [Gennari, 2002] it is worked out how fix-point reasoning can be applied to constraint propagation and modal logic in a general way.

Fix-point reasoning bases on the data type of a complete partial order (CPO). We will mainly base our presentation on [Pepper and Hofstedt, 2006].

Definition 6.1 (complete partial order (CPO))**.** A *partially ordered set* (poset) is a set $\mathcal{A}$ with a reflexive, antisymmetric and transitive relation $\preceq$.

The operation $x \sqcup y$ (*join*) yields the smallest element $z \in \mathcal{A}$ such that $x \preceq z$ and $y \preceq z$. The generalization for sets $\mathcal{X}$ is obvious. The element $\sqcup\mathcal{X}$ is called the *supremum* of $\mathcal{X}$.

An *ascending chain* of a poset $(\mathcal{A}, \preceq)$ is a subset $\mathcal{K} \subseteq \mathcal{A}$ consisting of strictly ascending elements: $\mathcal{K} = \{x_0 \prec x_1 \prec x_2 \prec \dots\}$

A poset $\mathcal{A}$ is *complete* if every ascending chain $\mathcal{K} \subseteq \mathcal{A}$ has a supremum $\sqcup\mathcal{K} \in \mathcal{A}$.

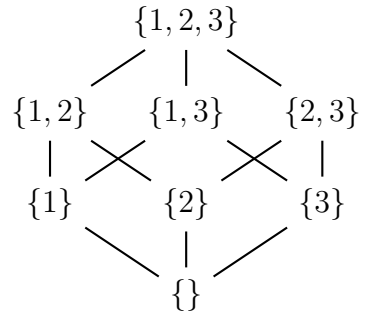


Figure 6.1.: CPO of the example domain $\{1, 2, 3\}$.

Such a *complete poset* (CPO) specifically includes a smallest element $\bot$ that is designated by the supremum of the empty chain: $\bot = \sqcup\emptyset$.

Figure 6.1 shows an example of a specific set CPO. The ordering relation $\preceq$ on the CPO elements is $\subseteq$. Since the domain of an integer finite-domain variable is essentially a set, we will base our reasoning on these data types by exploiting the CPO properties. CPOs allow the working with fix-points. The basic definition of such a fix-point is rather simple.

Definition 6.2 (fix-point)**.** An element $x \in \mathcal{A}$ is a *fix-point* of a function $f : \mathcal{A} \to \mathcal{A}$ if $f(x) = x$.

We call x' the *least fix-point* if $x' \preceq y$ for all other fix-points $y \in \mathcal{A}$.

The *least conditional fix-point* x'_w is defined by the least x'_w such that $w \preceq x'_w \wedge x'_w = f(x'_w)$. Thus x'_w is the smallest fix-point of f comprising w.

Since fix-points are not uniquely defined we need to establish a method to define one of the possibly many fix-points as the solution. The CPO properties in conjunction with monotonic functions ensure that one unique least fix-point exists.

Definition 6.3 (monotone, continuous, inflationary functions)**.** Let $(\mathcal{A}, \preceq_A)$ and $(\mathcal{B}, \preceq_B)$ be CPOs. We define the following notions:

- A function $f : \mathcal{A} \to \mathcal{B}$ is called *monotone* if it preserves the order of its arguments: $x \preceq_A y \Rightarrow f(x) \preceq_B f(y)$.
- A function $f : \mathcal{A} \to \mathcal{B}$ is called *continuous* if it preserves the suprema: $f(\sqcup\mathcal{K}) = \sqcup(f * \mathcal{K})$, where $*$ is the usual map function and $\mathcal{K}$ is an ascending chain. Particularly, every continuous function is also monotone.
- A function $f : \mathcal{A} \to \mathcal{A}$ is called *inflationary* iff $x \preceq_A f(x)$.

Continuous functions provide us with a constructive scheme to compute the least fix-point starting with $\bot$.

Theorem 6.1 (Kleene's fix-point theorem [Kleene, 1952])**.** Every continuous function f defined on a CPO has a least fix-point x'. This fix-point is determined by the least upper bound of the ascending Kleene chain of f, that is $\bot \preceq f(\bot) \preceq f^2(\bot) \preceq f^3(\bot) \preceq \ldots$, i.e. $x' = \sqcup\{\bot, f(\bot), f^2(\bot), \ldots\}$.

For conditional fix-points we require the inflationary property of the function. For a conditional starting point w we obtain the Kleene chain as $x'_w = \sqcup\{w, f(w), f^2(w), \ldots\}$, with the prerequisite that $w \preceq f(w)$ holds.

The fix-point computation scheme alone does not guaranty a terminating computation of the sought fix-point since the ascending Kleene chain is potentially infinite. In the next section we will consider the sufficient conditions for a terminating computation and how they apply to the constraint solving process.

6.2. Termination Foundations and Requirements

Repeated application of reduction functions (propagators) is the underlying method for solution derivation in the solver system. The fix-point iteration occurs mainly at three points within the solution process:

1. The overall propagator iteration, which is applied at every search level until no further changes can be inferred by the different propagators (cf. Section 4). It is executed for every node in the search tree.

2. The heuristic variable and value selection is a higher-level fix-point iteration. It is applied, if propagator iteration alone does not lead to a singleton domain. It operates on the same CPO and is interlocked with the propagator fix-point derivation. The combined application of these two fix-point computations until a singleton domain is reached represents one path in the search tree.

3. Inside idempotent propagators that establish a fix-point solution on the domains of participating variables.

The property of idempotence of a propagator requires that an additional run will not lead to any further domain changes, i.e. the propagator has reached a local fix-point itself. The algorithm for computing the fix-point is highly propagator specific and thus not of concern to our overall termination consideration. In fact, propagator idempotence is only an optimization to ease and speed up propagator

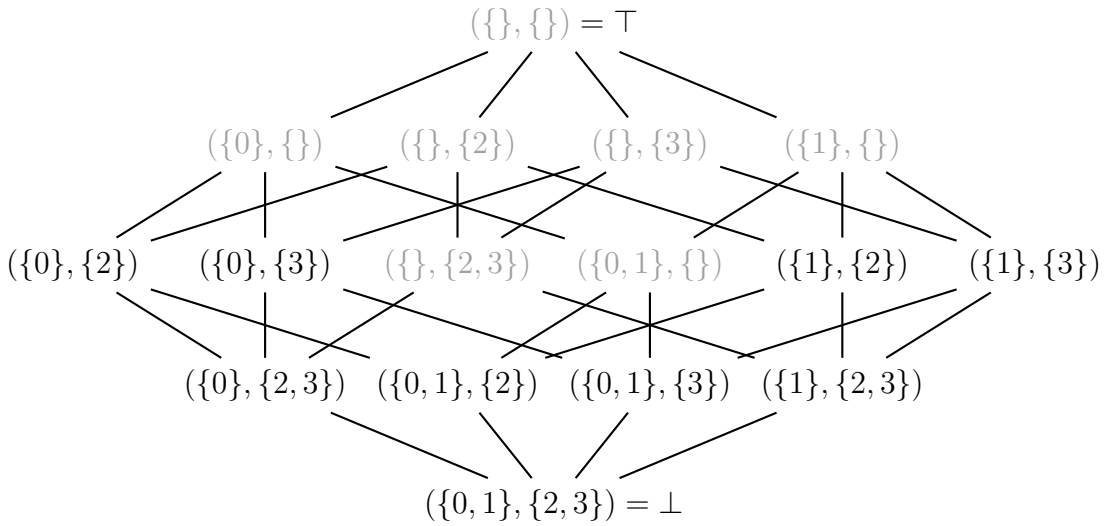


Figure 6.2.: Lattice for a two-Variable Domain-Tuple $(\{0,1\}, \{2,3\})$ with $\preceq \equiv \supseteq$ holding for every tuple item.

scheduling and is not required for an overall terminating propagator iteration. We will therefore focus on propagator iteration and the overall search process.

Since CPOs allow the reasoning about fix-points we have to discuss that the underlying data structure actually fulfills the CPO criteria. The fundamental data structure for the finite domain solver is the domain data type that is associated to every variable. We will treat the combined domain of a constraint problem with n variables as an n-tuple where tuple item i is the domain of variable x_i ($i \in \{1, \ldots, n\}$). Figure 6.2 is an example structure for a two-variable constraint system on integer finite domains. The constraint system consists of two variables x, y that are restricted to $x \in \{0,1\}$, $y \in \{2,3\}$ as initially allowed values.

The resulting structure fulfills the CPO properties if the tuple-elements are a CPO themselves. The functions $\preceq$ and $\sqcup$ are defined as follows. Let $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ be tuples, where the a_i and b_i are CPOs themselves. Then

$$(a_1, \ldots, a_n) \preceq (b_1, \ldots, b_n) = \bigwedge_{i=\{1,\ldots,n\}} a_i \preceq b_i$$

and

$$(a_1, \ldots, a_n) \sqcup (b_1, \ldots, b_n) = (a_1 \sqcup b_1, \ldots, a_n \sqcup b_n)$$

define the respective structure operations.

In our example we have turned the element order of the set-CPO as exemplified in Figure 6.1 upside-down by inverting the $\preceq$-relation, specifically we set $\preceq \equiv \supseteq$ for the ordering relation on every tuple element and define $\sqcup \equiv \cap$ as join. This is possible, because our value sets are always finite and we are thus always able to identify a bottom element. The order is then used element-wise for every item of the n-tuple. We have inverted the relation since CSPs start with fully populated domains that may contain conflicting values which are then step-wise reduced to the singleton (or empty) domain. The resulting structure represents a *lattice*.

Definition 6.4 (lattice)**.** A poset $(\mathcal{A}, \preceq)$ is called a *lattice* if for all elements $x, y \in \mathcal{A}$ the set $\{x, y\}$ has both a supremum (denoted by the binary *join* operation $x \sqcup y$) and an infimum (denoted by the binary *meet* operation $x \sqcap y$) in $\mathcal{A}$. Such a lattice also has a greatest and least element $\top$ and $\bot$.

A lattice is *complete* if all of its subsets have both a join and a meet in $(\mathcal{A}, \preceq)$. Every finite lattice is complete.

Within the actual solver implementation all the lattice elements that contain at least one empty set item (even though each element individually represents a possible fix-point)[1] are collapsed into a single $\top$-element as this constitutes a no-solution situation anyway and the solving process is thus aborted immediately for performance considerations[2].

The following theorem in conjunction with Theorem 6.1 enables the fix-point computation on lattices. It ensures the existence of the sought fix-point within the lattice structure.

Theorem 6.2 (Knaster-Tarski fix-point theorem [Tarski, 1955])**.** Let $f : \mathcal{A} \to \mathcal{A}$ be a monotonic function on a complete lattice $(\mathcal{A}, \preceq)$. Then f has a least fix-point x' and a greatest fix-point $\overline{x}$[3].

Initially every variable domain must contain a finite set of elements. The reduction functions reduce them to singleton or empty domains. This induces that every ascending chain is finite. Theorem 6.1 describes the iteration steps necessary for the fix-point computation by repeatedly applying a continuous function f starting with $\bot$ as initial argument.

[1] These elements are colored gray in Figure 6.2.

[2] From a theoretical point of view we simply compute a fix-point and check afterwards if any domain was reduced to $\emptyset$. The actual solver interleaves this check with the fix-point computation for performance reasons since this way it is often possible to abort a failing computation branch early in the process.

[3] However, $\overline{x}$ is not of interest for our problem domain.

Since we do not only want to apply a single propagator we define $f := p_1 \circ p_2 \circ \cdots \circ p_k$ for k propagators where $\circ$ is the usual function composition, i.e. $(f \circ g)(x) = f(g(x))$. If every p_i ($i \in \{1, \ldots, k\}$) is continuous the resulting function f is continuous. Note that semantically this computes the conjunction of all k propagators $p_1 \wedge \cdots \wedge p_k$ if f is iterated until the least fix-point is reached.

Lemma 6.1 (Continuity of combined functions)**.** If f and g are two continuous functions then their combination $f \circ g$ is continuous as well.

Proof. We want to show that if f and g are continuous then their composition $f \circ g$ is continuous as well. From the continuity assumption we can expect $f(\sqcup\mathcal{K}) = \sqcup(f * \mathcal{K})$ to hold for both functions f and g:

$$\begin{array}{lcll} f \circ g(\sqcup\mathcal{K}) & = & & \text{by definition of } \circ \\ f(g(\sqcup\mathcal{K})) & = & f(\sqcup(g * \mathcal{K})) & \text{by continuity of } g \\ & = & \sqcup(f * (g * \mathcal{K})) & \text{by continuity of } f \\ & = & \sqcup(f \circ g * \mathcal{K}) & \text{by definition of function composition and semantics of the map operator } (*) \end{array}$$

Thus we can conclude that the composition $f \circ g$ is of two continuous functions f and g is also continuous. □

Since the carrier sets of our CPOs are finite we are able to make use of the following property concerning the ascending chains within the data structure.

Lemma 6.2 (Least upper bound in finite chains)**.** The least upper bound of a finite ascending chain is its unique largest element.

Proof. Consider a finite ascending chain $x_0 \preceq x_1 \preceq \cdots \preceq x_n$. Obviously $x_i \preceq x_n$ holds for every x_i ($i \in \{0, 1, \ldots, n\}$). Suppose, there is another, smaller least upper bound x_l. This implies that $\sqcup\{x_0, \ldots, x_n\} = x_l$ and therefore $x_n \preceq x_l$. Due to the antisymmetry of the ordering relation we conclude that $x_l = x_n$ follows from $x_l \preceq x_n \wedge x_n \preceq x_l$. □

Continuity is the underlying concept and precondition of Theorem 6.1. However, the property of monotonicity is usually more intuitive and easier to check. Fortunately, within our restricted setting of complete lattices on finite sets as underlying data structure it is possible to show that monotonicity is the only fundamental property that also implies continuity.

Lemma 6.3 (Precondition such that monotonicity implies continuity)**.** Let $f : \mathcal{A} \to \mathcal{A}$ and let $\mathcal{A}$ be finite, then monotonicity of f also implies continuity of f.

Proof. Finiteness of $\mathcal{A}$ implies that there are only finite chains $x_0 \preceq x_1 \preceq \cdots \preceq x_n$, $\mathcal{X} = \{x_0, \ldots, x_n\} \subseteq \mathcal{A}$. Let $f : \mathcal{A} \to \mathcal{A}$ be a monotonic function, then clearly $f(x_0) \preceq f(x_1) \preceq \cdots \preceq f(x_n)$ holds.

From Lemma 6.2 we know that $\sqcup\mathcal{X} = x_n$ holds, and thus $f(\sqcup\mathcal{X}) = f(x_n) = \sqcup(f * \mathcal{X})$. □

Using the above reasoning we are able to support the necessary propagator conditions as stated in Chapter 4, i.e. monotonicity and inflation. The underlying data structure is a CPO or more specifically a lattice. The solving process always starts at the $\bot$ element of this structure. The function f is the combination of the subscribed propagators for the current constraint problem. A fundamental property for the application of Theorem 6.1 is the *continuity* of f. Lemma 6.3 ensured that this condition is already ensured by monotonicity of f.

In [Gennari, 2002] inflation is assumed as a generally necessary property for arbitrary domains. Since we start the Kleene chain at $\bot$ and are not searching a least *conditional* fix-point there is no need for this additional function property [Cai and Paige, 1989]. Moreover, since we start at $\bot$ (the smallest lattice element) there are only two possible options for the application of f during an iteration step:

1. We are already at a fix-point and no changes occur, or
2. We climb the ascending chain and the result is a greater element of the lattice. Since we start at the smallest element and f is monotonic, the only possible operation is the removal of values from the tuple-sets.

This implies that a monotone propagator in our setting is limited to be *contracting* if it makes any domain changes at all, i.e. it is only allowed to remove values from a domain without violating the fix-point theoretical foundations. That also implies that $x \preceq f(x)$ holds and thus the inflationary property of propagators always holds as well.

This ability to compute conditional fix-points and thus the requirement for an inflationary function is not explicitly needed. However, since we just argued that the sole operation for propagators on domains is value removal, this additional property follows as well (but is not essential for our termination argumentation).

We may thus conclude the necessary propagator properties based on fix-point theory that ensure a terminating solving process. Since the underlying complete lattice data type is finite, all ascending chains are finite as well. Monotonous propagator functions will compute a least fix-point by using the Kleene chain starting at $\bot$. We have also shown that only contracting propagators fulfill these conditions, i.e. a propagator may not add values to a domain during the pruning process.

This fix-point finding is repeated for every visited node of the search tree. Every time a new node is created one variable domain will have changed due to the variable and value selection heuristic. The new node is derived from a level where the conjunction of all propagators has already computed its fix-point. The resulting domain as computed by the selection heuristic represents a new CPO with a new $\bot$ to start the Kleene chain at, eventually reaching $\top$ (and subsequently signaling a failure) or deriving the singleton domain.

Node creation consists of the following two steps:

1. First, a least one variable with a non-singleton domain is chosen,
2. Next, new nodes are created by selecting one or more domain values of the variable for each node.

The second step is either achieved by simple labeling (a node is created for each domain value and thus the selected variable has a singleton domain in each node) or by more sophisticated value selection heuristics (cf. Chapter 5).

Since domains are finite this process is bound to terminate eventually as well.

6.2.1. Optimization using Micro-Steps

In [Pepper and Hofstedt, 2006; Cai and Paige, 1989] the concept of micro steps and work sets is introduced to allow for a faster convergence. The underlying principle is a generalization of the fix-point chain $\{w, f(w), f^2(w), \ldots, f^i(w), f^{i+1}(w), \ldots\}$. The main steps $f^i(w) \mapsto f^{i+1}(w)$ may often be divided into several sub-steps (or micro steps) such that $\{\ldots, f^i(w), s_{i_1}, s_{i_2}, \ldots, s_{i_n}, f^{i+1}(w)\}$. Furthermore, the micro steps are not necessarily synchronized with the main steps, therefore we are regarding sequences of type $\{w = s_0, s_1, \ldots, s_i, s_{i+1}, \ldots\}$. It is possible to show ([Pepper and Hofstedt, 2006]) that for finite fix-points x_w of such sequences $x_w = x'_w$ holds, i.e. one derives the same fix-point either by iterating on the micro steps or by computing on the standard (macro steps) Kleene chain. In [Cai and Paige, 1989; Pepper and Hofstedt, 2006] the following corollary is deduced.

Corollary 6.1 (Corollary 10.2. in [Pepper and Hofstedt, 2006])**.** If f is of the special form

$$f(x) = f_1(x) \sqcup \cdots \sqcup f_k(x)$$

then the s_i can be derived from a special work set, where only one of the f_j functions is used:

$$\Delta_j(x, f) = \{s \,|\, x \prec s \preceq f_j(x)\}$$

That means that we compute a fix-point by applying several sub-functions in a step-wise manner rather than iterating on a complex function f. By reusing the result of the previous step the iteration process often converges a lot faster.

An unnoticed use of this property has already been applied in the previous section. By combining the individual propagators to $f = p_1 \circ \cdots \circ p_k$ each p_i ($i \in \{1, \ldots, k\}$) implicitly makes use of the computation result of the previous propagator due to the definition of $\circ$. The more general definition $f = p_1 \wedge \cdots \wedge p_k$ would require the computation of individual results each from the same argument domain and a subsequent intersection (by definition of $\wedge$) to calculate the result of f. This is computationally a lot more expensive. But we can still do better by making even better use of Corollary 6.1.

The concept of work sets as introduced in Corollary 6.1 is applicable if f is of the form $f(x) = f_1(x) \sqcup \cdots \sqcup f_k(x)$. This corresponds to our conjunctive form above because for the integer set CPO we define $\sqcup \equiv \cap$ which corresponds to the conjunction of the individual f_i or more specifically our propagators p_i. Two operations are required:

- $\Delta_i(x, f) = \{d \,|\, x \prec x \oplus d \preceq f_i(x)\}$, the computation of the work set.
- $\oplus$ the incremental advance on the underlying CPO data type with the help of a work set element.

We try two achieve two different levels of optimization: 1. the already established reuse of computation results of other propagator applications, and 2. the specific rerun of propagators affected by domain changes. The managed work set will be the set of propagator functions queued for application. The operations need to be defined as follows:

- $\oplus$ is the propagator application to a domain tuple
- all propagators that implement constraints on a changed domain are added to the work set.

Using the syntax of [Pepper and Hofstedt, 2006] we obtain Algorithm 6.1.

In the algorithm illustration we describe the conjunction of propagators p_i as a vector of functions and denote it as f. There are two versions of the *fix-point* function. The first one expects no initial work set but derives it as the set of all functions in the propagator vector. The second variant is supplied with a work set argument[4]. For *iterate* two cases exist: either the work set is empty, then

[4]This variant may be used after a heuristic choice was made where only some of the propagators may need to be scheduled. This variant is only an optimization and not necessary for the general formulation.

Algorithm 6.1: Fix-point Iteration Algorithm using Micro-Steps.

```
1  FUN fix-point : Seq (α → α) → α                                  // fix-point from ⊥
2  DEF fix-point(f) = iterate(f)(⊥, asSet(f))                       // work set derived from f
3  FUN fix-point : Seq (α → α) → Set β → α                          // fix-point with initialized
4  DEF fix-point(f)(w) = iterate(f)(⊥, w)                           // work set
5  FUN iterate : Seq (α → α) → α × Set β → α
6  DEF iterate(f)(s, w) =
7  |  IF w = ∅ THEN
8  |  |  s
9  |  ELSE
10 |  |  LET
11 |  |  |  p = arb(w)                                             // choose propagator from work set
12 |  |  |  s' = p(s)                                              // apply it on s, i.e. s' = s ⊕ p
13 |  |  |  w' = w ∪ attached-props(s, s', f)                      // expand work set
14 |  |  IN
15 |  |  |  iterate(f)(s', w')                                     // continue fix-point computation
16 |
```

the computed domain is returned, or a propagator is removed from the work set. Afterwards it is applied to the domain data type. Finally all propagators dependent on a changed domain are added to the work set. This is done by using *attached-props*. The function *attached-props* returns a set that contains all propagators that are defined to operate on a domain that is changed between s and s'. Looking closely, this corresponds to the AC-3 algorithm as displayed in Section 4.1, but here it is derived from the foundations as laid out by fix-point theory.

6.3. Uniqueness of Solutions

Having elaborated on the conditions that influence the termination of the solution process we also want to be able to reason about the particular (of possibly many) solution that is derived. In general a CSP may have no, one or finitely many solutions. In the case of no or one solution the derived one is obviously unique. If there exists more than one solution we obtain the following alternatives. Basicly, a user may choose between three different modes:

1. find all solutions,
2. find the/a best solution, or

3. find one (or the first n) solution(s).

Search modes 1. and 2. must visit the whole search tree to prove the completeness (for all solutions) or the optimality of the solution (for the second case). The set of computed solutions for case 1. is unique and no other result is derivable. However, if the constraint that describes what is "best" does not apply to one single solution, the search heuristic again determines the particular solution computed.

Search variant 3. is the general case. It is obvious that the solution that is finally derived by the solver is determined by the (heuristically) chosen variable and the chosen value at each choice point. If a heuristic does not use randomness the result is reliably repeatable. It is important to keep in mind that fix-point iteration within the nodes does not influence which one of a number of possible solutions is computed. This is entirely influenced by the search part.

7. Performance Evaluation

To support and evaluate our overall design and implementation decisions we compared our CLFD system to other systems falling into two classes: a statically typed and fully compile-time resolved solver system and another class of a dynamic, run-time-typed system. For the first class we chose GECODE as competing solver system, because it is currently the fastest and most sophisticated solver available; for the second class of dynamic systems we compare to SCREAMER as a well-known and established solver implemented in a dynamically typed language (cf. Section 2.5).

In the next section we will first introduce the statistical model which forms the base of our comparison of the benchmark data. Next we a giving an overview of the different benchmark problems in Section 7.2. Finally, Section 7.3 presents the actual benchmark data and analyzes the results.

7.1. The Statistical Model

When measuring the performance of a certain system it is usually assumed that external events do not influence the obtained results. However, in real world (computer) systems there are a number of factors that could perturb the measurements. For example, real time events such as interrupts to service network interfaces or clocks, as well as intermittent events of the operating system's memory management such as cache misses or page faults all influence the results. Such non-deterministic behavior is even more aggravated by the inherent behavior of dynamic run-time systems such as the Java Virtual Machine (JVM) or, in our case, the underlying Common Lisp run-time system. In [Georges *et al.*, 2007] it is shown that ignoring the non-deterministic influences in such systems may result in wrong or misleading conclusions. The JVM exhibits a number of such non-deterministic sources [Georges *et al.*, 2007]:

- run-time/just-in-time compilation
- just-in-time compilation bases on internal timers that guide the optimization process and may turn out different for every JVM start-up

- (self-adjusting) garbage collection
- on-demand code loading.

Similar sources of measurement perturbation can be found in a Common Lisp system, such as:

- on-the-fly generation and compilation of optimized accessor and dispatch functions in the Common Lisp Object System (CLOS)
- cache behavior/warm up (especially for the CLOS sub-system)
- garbage collection and its automatic tuning
- fluctuations due to different code alignment which may change for different Lisp system invocations[1]

In [Lilja, 2000; Georges *et al.*, 2007] a statistical model for handling and describing such *random errors* is presented and its influence on the result interpretation is discussed. We will next introduce the relevant parts of this model and later apply it to our benchmarking procedure and the result interpretation. It is of course still up to the benchmark designer to avoid any *systematic errors* which introduce bias into the measurements. The statistical approach is only used to approach the unavoidable noise in the timing data. The model will use *confidence intervals* to quantify the repeatability of the measurements[2].

7.1.1. Confidence Intervals

A *confidence interval* is a kind of interval estimate for a population parameter. Statistical confidence intervals are used to find a range of values that has a given probability to include the actual value. We have to differentiate two situations: a large sample size (where large in statistics typically means $n \geq 30$); and the case for small samples sizes ($n < 30$).

Since we do not know our actual distribution mean x, but only have a sample of measurements, x is approximated by the sample mean $\overline{x}$.

[1] x264 Development diary (`http://x264dev.multimedia.cx/?p=51`): "For a reason we have yet to figure out (though we have much speculation), the Core 2 has this odd habit of execution times changing dramatically solely due to alignment of the code itself. That is, one could literally speed up a small segment of code just by inserting random numbers of `nops` before it until it got faster. This wasn't useful for optimization, as it was not only random but misaligning one set of code would hurt another set of code just as much."

[2] Note that this approach requires that the random errors are Gaussian distributed; [Lilja, 2000] (Section 4.3) argues why this is in general a reasonable assumption.

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{7.1}$$

If the x_i are independent and from the same population with mean μ and standard deviation σ, then central limit theory assures that for large n (i.e. $n \geq 30$) $\overline{x}$ is approximately Gaussian distributed with mean μ and standard deviation $\sigma/\sqrt{n}$. We thus can assume that the true value x that we are trying to measure is the population mean μ. The x_i sample values therefore occur with a probability that follows the Gaussian distribution.

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1} \tag{7.2}$$

Standard deviation s is computed from the sample variance $s = \sqrt{s^2}$. The *coefficient of variance* $s/\overline{x}$ normalizes the standard deviation and provides a dimensionless value that compares the relative size of the variation in the measurements to the mean.

We are looking for two values c_1 and c_2 such that the probability of the mean value being between these values is $1 - \alpha$:

$$\Pr[c_1 \leq x \leq c_2] = 1 - \alpha \tag{7.3}$$

The two values can thus be used to quantify the precision of our measurements. The interval $[c_1, c_2]$ is the *confidence interval* for the mean value $\overline{x}$, α is called the *significance level* and the number $1 - \alpha$ is the *confidence level*. Applying a normalization

$$z = \frac{\overline{x} - x}{\sigma/\sqrt{n}} \tag{7.4}$$

transforms $\overline{x}$ to follow the standard unit normal distribution (i.e. a Gaussian distribution with mean $\mu = 0$ and variance $\sigma^2 = 1$). According to the central limit theorem the sought interval borders are

$$c_1 = \overline{x} - z_{1-\alpha/2}\frac{s}{\sqrt{n}} \qquad c_2 = \overline{x} + z_{1-\alpha/2}\frac{s}{\sqrt{n}} \tag{7.5}$$

The value $z_{1-\alpha/2}$[3] can be obtained from a precomputed table. For a large sample size the actual variance σ^2 is approximated by the sample variance s^2. This allowed for the normalization transformation in Equation 7.4.

For small sample sizes, which in statistics is typically assumed as $n < 30$, the sample variance s^2 for several groups of measurements may vary significantly for

[3]This is the value of a standard unit normal distribution that has an area of $1 - \alpha/2$ to the left of $z_{1-\alpha/2}$.

each group. For this case statistics theory has shown that the normalization in Equation 7.4 does not follow the standard unit normal distribution but the Student's t-distribution with n degrees of freedom[4]. Thus, for determining the confidence interval for small n ($n < 30$), the bounds can be computed as

$$c_1 = \overline{x} - t_{1-\alpha/2;n-1}\frac{s}{\sqrt{n}} \qquad c_2 = \overline{x} + t_{1-\alpha/2;n-1}\frac{s}{\sqrt{n}} \tag{7.6}$$

Similar to Equation 7.5, $t_{1-\alpha/2;n-1}$ is the value from the t-distribution with $n-1$ degrees of freedom.

7.1.2. Comparing Two Alternatives

It is our ultimate goal to compare two systems with each other, or more specifically we want to compare our solver framework to another constraint solver system. For this goal confidence intervals are well suited. Note, that this approach only allows us to make predictions about the performance between two alternatives at the same time, it is not suited to make a comparison of all systems together at the same time.

We cannot guarantee a direct correspondence between pairs of measurements, especially because the number of runs may not be the same for each system. The measurements are therefore *uncorresponding*. Statistics has a well established approach for this situation.

First we need to calculate the *difference of the means* $\overline{x} = \overline{x}_1 - \overline{x}_2$. It can be shown that the standard deviation of this difference of mean values is the sum of the standard deviations of each set of measurements, weighted by the total number of measurements in each set [Fahrmeir *et al.*, 2007]:

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \tag{7.7}$$

In this way we gain an estimate of the difference of mean values of each set of measurements and an estimate of each standard deviation. By following the same derivation as in Section 7.1.1 we can calculate the confidence interval for the difference of the means by

$$c_1 = \overline{x} - t_{1-\alpha/2;n_{df}}s_x \qquad c_2 = \overline{x} + t_{1-\alpha/2;n_{df}}s_x \tag{7.8}$$

Note, that the necessary *number of degrees of freedom* for the evaluation of the t-distribution must be approximated by the following equation [Lilja, 2000]

[4]For large n the t-distribution approaches the Gaussian distribution [Lilja, 2000].

$$n_{df} = round\left(\frac{\left(\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}\right)^2}{\frac{\left(s_1^2/n_1\right)^2}{n_1-1}+\frac{\left(s_2^2/n_2\right)^2}{n_2-1}}\right) \tag{7.9}$$

We can now make use of these confidence intervals to analyze the statistical importance of the differences found in the measurements. If the confidence intervals enclose the number 0 then there is no statistical evidence that one system is better than the other and that all witnessed differences can be assumed to be the source of random fluctuations. This describes the case when the two confidence intervals of the compared measurements overlap.

7.2. Benchmark Problems

A number of constraint problems have been employed to test and analyze the solvers. Before focusing on the actual run-time data we will briefly illustrate the used problem examples.

Since SCREAMER does not contain any global constraint we added a primitive variant of the *all-different* constraint based on the SCREAMER$^+$-additions [White and Sleeman, 1998; White, 2000] to enable a better comparability of the result. However, since the realization of a similar algorithm in SCREAMER would be fairly involved (and was one of the initial pressing factors for a new solver design), it is only a very simple variant, though it performs still remarkably better than a respective number of binary $\neq$ relations, SCREAMER would otherwise normally use.

7.2.1. Sudoku

Sudoku [Garns, 1976] is a constraint problem *par excellence* that has also become popular in recent years as a pastime amusement. The base is a 9×9 grid sub-divided into 9 blocks of 3×3 cells. Some of the grid cells are populated with random numbers in the range $1\ldots9$. It is the goal to find the numbers in the range $1\ldots9$ for all other cells such that every row, column or block contains every number only once[5]. Figure 7.1 shows an initial Sudoku problem as well as the solved puzzle.

The benchmark first consists of solving two test problems, a simple and a hard one. A second test uses the suite from [Norvig, 2006] which consist of 95 different Sudoku puzzles. The benchmark includes both, problem construction and initialization, as well as the actual solving time.

[5]There exist also variations that use larger grids, e.g. a 12×12 field.

		8				6		
	4		9		2		5	
			6	4	8			
	3	9		2		1	7	
	1						3	
	8	5		1		2	6	
			2	8	7			
	6		1		4		8	
		2				5		

(a) The initial puzzle.

3	9	8	5	7	1	6	2	4
6	4	1	9	3	2	8	5	7
5	2	7	6	4	8	9	1	3
4	3	9	8	2	6	1	7	5
2	1	6	7	9	5	4	3	8
7	8	5	4	1	3	2	6	9
1	5	4	2	8	7	3	9	6
9	6	3	1	5	4	7	8	2
8	7	2	3	6	9	5	4	1

(b) The solved puzzle.

Figure 7.1.: A Sudoku puzzle.

The constraint model for a Sudoku puzzle is relatively simple: There is a variable for every cell of the puzzle field, each initialized to the integer domain $\{1, \ldots, 9\}$. Predetermined field numbers are of course initialized to their respective specified digit. Furthermore, we need 27 *all-different* constraints on the 9 variables in each respective row, column and 3×3 sub-field.

7.2.2. n-Queens

The well known and widely employed n-Queens[6] problems demands that n queens shall be placed on a $n \times n$ chess board in such a way that all the queens are in mutually friendly positions. Typically, the problem model is such that the queens are already numerated and thus each assigned to a different row, as it is obvious that no two queens can share the same row[7]. The constraint model has then only to account for the different column position of each queen and the mutually friendly specification. Figure 7.2 depicts one of the 92 possible solutions for the 8-Queens puzzle.

There are several ways to model this CSP. We have decided to test several ways to express the puzzle as to see whether the reference solvers can handle one model prominently better than to accidentally pick a low performing description.

The basic CSP model can be expressed as follows. This corresponds to Model A

[6]This problem has half-jokingly been called the un-avoidable benchmark for finite domain constraint solvers [Zoeteweij, 2005].

[7]Alternatively, one can of course swap the meaning of rows and columns here.

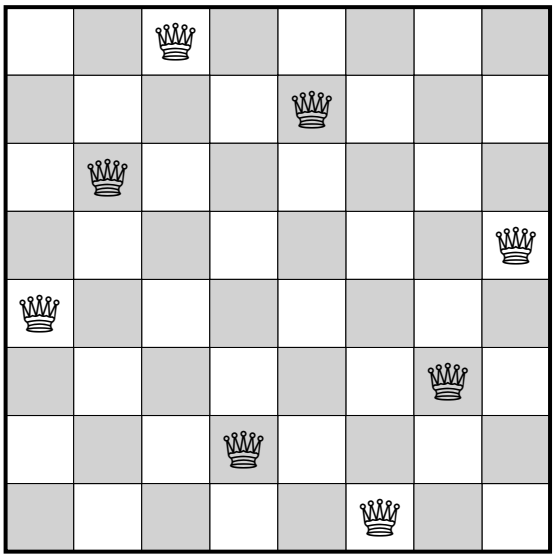


Figure 7.2.: A possible solution placement for the 8-queens problem.

of the different models below. The $\neq$-constraints disallow the occupation of fields in the same column and in the diagonals of some queen q_i.

$$q_1, \ldots, q_n \in \{1, \ldots, n\} \quad \wedge$$
$$\bigwedge_{1 \leq i < j \leq n} (q_i \neq q_j \wedge q_i - j \neq q_j - i \wedge q_i - i \neq q_j - j)$$

Model A The most obvious model is to express (and thus by using $\neq$ to disallow) the queens' mutual threats by an according number of $\neq$ constraints for the occupied board fields and theirs respective columns and diagonals. This leads to a relatively high number of constraints but a more influencing factor is that all relations are between two queens only, thus impeding any form of global reasoning on the interrelation of all queens together.

Model B We replace the $\frac{3}{2}n(n+1)$ different $\neq$-constraints of the previous model with three *all-different* constraints plus additional offset rules to express the forbidden threats via rows, columns and diagonals[8]. This is very similar to Model A but the application of the global all-different constraint allows a more sophisticated analysis of the coupling of (n-ary) inter-queens relations.

[8]Depending on the solver the offset rules can be expressed inside the *all-different* constraint or have to be added as additional constraints with respective auxiliary variables.

Model C This model corresponds the CSP description used as example in the introduction of the SCREAMER constraint solver of [Siskind and McAllester, 1993a]. It uses a closure on a predicate to check for the validity of a configuration.

This style of programming is very close to the common style of (functional) Lisp programming and thus feels very natural. However, since function application is only possible on fully instantiated variables, there is no way of large scale domain pruning. SCREAMER and CLFD both employ residuation style approach for this kind of propagator (cf. Section 4.2.4),

This generic higher-order function based model cannot be realized within GECODE without greater effort due to the lack of closures in C++.

Model D This fourth model uses a global constraint specific to the n-Queens problem [Régin, 2003]. The filtering algorithm is already able to remove values from related queens when the domain is reduced to two or three possible values, thus should be able to yield better reduction and earlier detection of conflicts.

7.2.3. Send-More-Money

A simple but well-known combinatorial crypto-arithmetic problem where the equation $SEND + MORE = MONEY$ has to be solved. Each letter represents a variable v with $v \in \{0, \ldots, 9\}$. There are two variants, a first one with only a single solution where $S > 0 \wedge M > 0$ is an additional required constraint, and a second one with 25 solutions where leading zeros are acceptable. Both variants make an exhaustive search through the search tree to prove that the discovered solution(s) is/are really the only one(s).

7.2.4. Kyoto

Find a base n such that the alphanumeric equation $KYOTO + KYOTO + KYOTO = TOKYO$ has a solution in the base-n number system [Dubrovsky and Shvetsov, 1995; Apt and Zoeteweij, 2006]. The problem is modeled with four variables K, Y, O, T and an additional variable N for the base. There are four constraints to express that N is the largest of all variables, i.e. $N > K, Y, O, T$, and one *all-different* constraint on the expression variables. Finally, the CSP includes one equality relation for the sum expression. The complete model can thus be expressed as follows:

$$
\begin{aligned}
& N > K, Y, O, T \quad \wedge \quad \textit{all-different}(K, Y, O, T) \quad \wedge \\
& N, K, Y, O, T \in \{0, \ldots, \textit{max-base}\} \quad \wedge \quad K, T > 0 \quad \wedge \\
& N^4K + N^3Y + N^2O + NT + O \\
+ & N^4K + N^3Y + N^2O + NT + O \\
+ & N^4K + N^3Y + N^2O + NT + O \\
= & N^4T + N^3O + N^2K + NY + O
\end{aligned}
$$

Reducing the sum to $3(N^4K + N^3Y + N^2O + NT + O)$ yields much increased performance during solution search since the number of multiplication and sum constraints is significantly reduced. For better comparability we have made this optimization for every solver by hand since the expression analyzer of CLFD will do it automatically and thus would lead to a hidden advantage.

The benchmark problem searches for solutions for bases in the range of *max-base* = 50. *Note*: The equation is only satisfiable in the number system of base 9, where there exist four different allocations for the equality relation.

7.2.5. Pythagorean Triples

The goal is to find all triples (a, b, c) such that the Pythagorean equation $a^2+b^2 = c^2$ is fulfilled. The domain of a, b, c is $\{1, \ldots,$n$\}$ for each variable where $n = 100$ is chosen and thus directly influences the size of the search space.

7.2.6. Golomb-Ruler

A Golomb ruler is a set of marks at integer positions such that no two pair of ticks are the same distance apart [Dewdney. Alexander K., 1985; Gent and Walsh, 1999]. The goal is to find a shortest ruler for a given order, e.g. for the order four the ruler $\langle 0, 1, 4, 6\rangle$ of length six is the shortest one. The ticks are enforced to be in increasing order. We use a branch-and-bound search to fulfill the optimization criteria. Apart from the required arithmetic criteria the problem description uses a bounds-consistent *all-different* propagator. [Smith *et al.*, 1999] presents an in-depth analysis of different approaches to model this problem.

7.2.7. Sum-Product

A problem involving very large domains from [Benhamou and Older, 1997]. The benchmark example uses $n = 14$ for its analysis. Notice, that this makes use of

large integer support because the numbers in the expressions get too large to be stored in a normal CPU register.

$$\begin{aligned} x_1, \ldots, x_n \in \{1, \ldots, n\} \quad & \wedge \\ c_1 \in \{1\}, c_2 \in \{2\}, \ldots, c_n \in \{n\} \quad & \wedge \\ x_1 + \cdots + x_n = c_1 + \cdots + c_n \quad & \wedge \\ x_1 * \cdots * x_n = c_1 * \cdots * c_n \quad & \wedge \\ x_1 \leq x_2 \leq \cdots \leq x_n & \end{aligned}$$

7.2.8. Cubes

This problem from [Apt and Zoeteweij, 2006] requires to find all distinct numbers $1 \leq n \leq 100\,000$ that are a sum of four different cubes. For example

$$1^3 + 2^3 + 3^3 + 4^3 = 100.$$

The model is quite simple and straight forward:

$$\begin{aligned} n \in \{1, \ldots, 100\,000\} \quad & \wedge \\ x_1, x_2, x_3, x_4 \in \mathbb{Z} \quad & \wedge \\ 0 < x_1 < x_2 < x_3 < x_4 < n + 1 \quad & \wedge \\ x_1^3 + x_2^3 + x_3^3 + x_4^3 = n & \end{aligned}$$

We impose an order on the variables to avoid rotated variants of a solution.

7.2.9. Fractions

This arithmetic task is a well-known benchmark for Prolog systems. The whole problem can be described as follows.

$$\begin{aligned} A, B, C, D, E, F, G, H, I \in \{1, \ldots, 9\} \quad & \wedge \\ \textit{all-different}\ \{A, B, C, D, E, F, G, H, I\} \quad & \wedge \\ \frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1 \quad & \wedge \\ \frac{A}{BC} \geq \frac{D}{EF} \geq \frac{G}{HI} \quad & \wedge \\ 3\frac{G}{HI} \leq 1 \leq 3\frac{A}{BC} & \end{aligned}$$

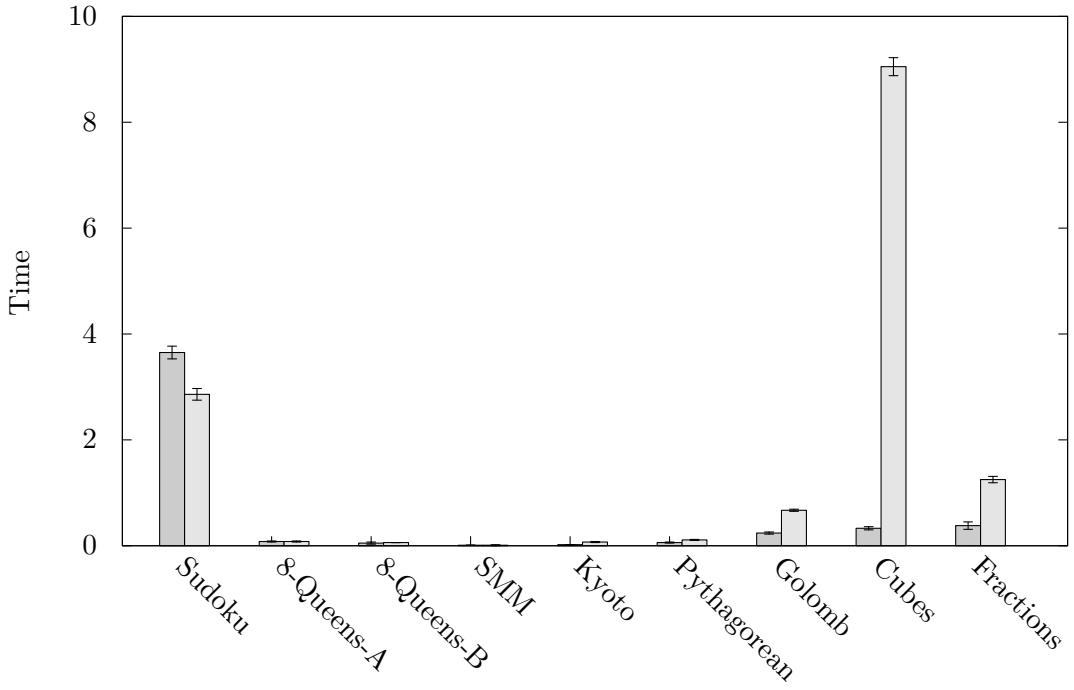


Figure 7.3.: Visualization of the performance comparison of Gecode vs. CLFD.

The first two constraint initialize the variables and require their distinctness. Next follows the actual problem expression. The last two constraints are for symmetry removal and redundancy to aid the domain pruning.

7.3. Evaluation

We ran the above benchmark problems for our test solver systems and applied the discussed statistical methods. We present our data first as a histogram to give a graphical illustration of the performance samples. The exact performance data is summarized in a table.

The values c_1 and c_2 are the lower and upper bound of the confidence interval of the of the difference of the means as defined in Equation 7.7. Remember, that this is interesting because if this confidence interval includes 0, we can conclude that for our chosen confidence level of 0.95, there is no statistically significant difference between the two sets of measurements.

Our first comparison is between Gecode and CLFD. The data is summarized in Figure 7.3 and Table 7.1, all times are in seconds.

Keep in mind that Gecode is a heavily optimized solver system with many

	Sudoku	8-Queens-A	8-Queens-B	8-Queens-C	8-Queens-D	SMM
Gecode	3.65	0.08	0.05	-	-	0.01
CLFD	2.86	0.08	0.06	0.08	0.06	0.01
c_1	0.69	-0.02	-0.01	-	-	-0.01
c_2	0.89	0.01	0.01	-	-	0.01

	Kyoto	Pythagorean	Golomb	Sum-Product	Cubes	Fractions
Gecode	0.02	0.06	0.24	-	0.33	0.38
CLFD	0.07	0.11	0.67	1.85	9.05	1.25
c_1	-0.06	-0.06	-0.45	-	-8.83	-0.93
c_2	-0.05	-0.04	-0.41	-	-8.60	-0.81

Table 7.1.: Detailed performance data for the comparison of Gecode and CLFD.

man-years of development and it is implemented in a statically typed and compiled language. However, for many problems we compare quite favorably. In the first problem set — Sudoku — there is a great number of Sudoku problems that are created and instantiated at run-time and are not pre-compiled into code. Thus the static features of the C++ compiler can not be applied and thus we gain an advantage in this case.

Overall, it appears that the limitation of Gecode domains to machine integers gives it an advantage in the arithmetic problems. But it also means that the Sum-Product problem results in a domain overflow and thus is too large for Gecode. It is not clear why the Cubes instance is so much slower for our systems. The most probable cause is a more specialized and better optimized propagator in the Gecode system. Queens-C and Queens-D cannot be realized in Gecode without greater effort and because of the lack of higher order functions in C++ and thus have not been compared.

Our second comparison is with the Screamer system that is more similar to our approach as it also bases on the dynamic language Common Lisp and works on problems that may contain large integers. We set out to integrate methods for greater solving efficiency and better modularity so these effort should reflect in our benchmark data.

Figure 7.4 presents the graphical histogram and Table 7.2 summarizes the exact performance values.

Our system performed better in almost all test cases with statistical evidence

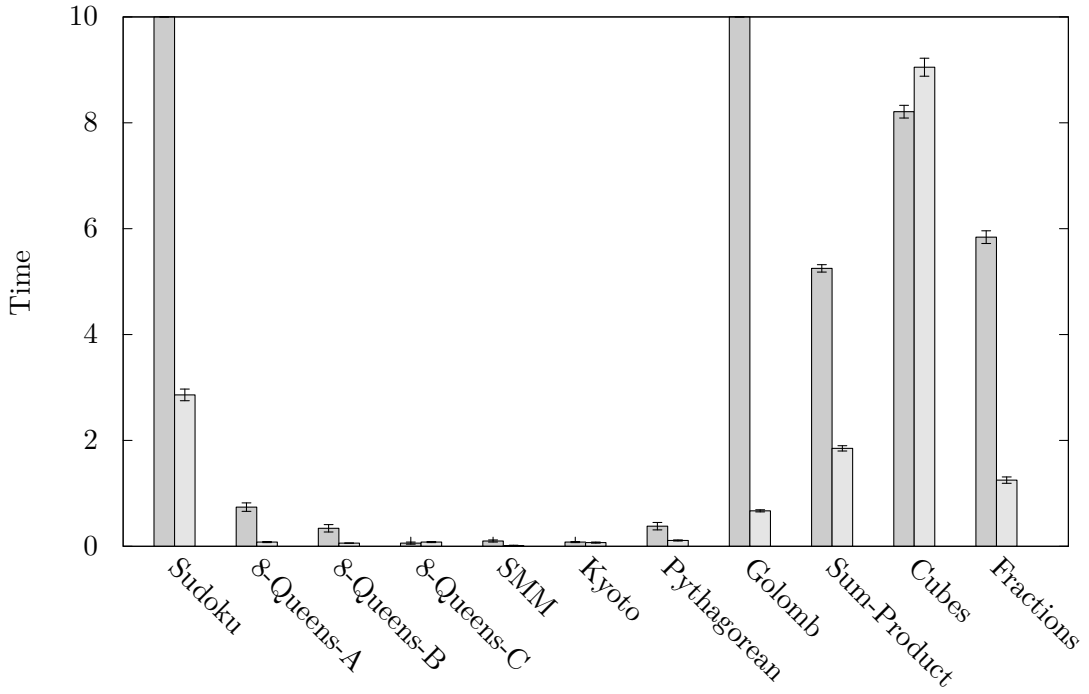


Figure 7.4.: Visualization of the performance comparison of Screamer vs. CLFD.

	Sudoku	Queens-A	Queens-B	Queens-C	Queens-D	SMM
Screamer	> 5min	0.74	0.34	0.06	-	0.10
CLFD	2.86	0.08	0.06	0.08	0.06	0.01
c1	-	0.60	0.24	-0.03	-	0.08
c2	-	0.71	0.33	0.00	-	0.10

	Kyoto	Pythagorean	Golomb	Sum-Product	Cubes	Fractions
Screamer	0.08	0.38	> 5min	5.25	8.21	5.84
CLFD	0.07	0.11	0.67	1.85	9.05	1.25
c_1	0.00	0.22	-	3.34	-0.96	4.50
c_2	0.02	0.32	-	3.46	-0.70	4.67

Table 7.2.: Detailed performance data for the comparison of Screamer and CLFD.

for an actual advantage. For two problems, Sudoku and Golomb, the benchmark has been aborted after 5 minutes in the case of Screamer because it was obvious that no solution would be found within a reasonable amount of time.

Overall, it has been shown that our design compares well to an established and heavily optimized system even though we are at a disadvantage due to the dynamic features at the language level. Secondly, our framework shows a significant advantage both in modularity, extensibility and performance when compared to the established Screamer system that shares the dynamic implementation foundations.

8. Integration Environment

The access to convenient programming and description of complex problems in a high-level and abstract way is the key point for the increasing success of declarative programming languages. Solver libraries such a ILOG SOLVER or CHOCO require the user to tediously create and interconnect instances of library classes to define constraints problems. The OPL language [Van Hentenryck and Michel, 2005] which hides all the C++ interface details of the SOLVER library to enable the comfortable definition of complex constraint problems in a declarative way, has since its introduction evolved into an extensive development environment. This demonstrates the importance of providing a simplified high-level interface for user interaction to abstract from most implementation details [Hofstedt, 2010].

In CLFD we realize such a high-level programming interface in two ways: First, there is a layer to describe constraint problems in a mostly declarative way. An initial design of the constraint interface abstraction has been introduced in [Frank and Hofstedt, 2005]. Since our framework is open for additions and enhancements, we aim to provide ways to simplify the addition of new constraint propagators, search goals and heuristic functions which at the same time ensure the compliance with the protocol requirements[1].

8.1. System Definition

At first, we will have a look at the definition of constraint problems. Figure 8.1 shows a simple example definition illustrating the use of the *stand-alone interface*. This interface acts as a batch processor where a problem is defined and a list of solutions is returned.

The constraint system is defined by lines 2–6. The resulting initial configuration is bound to the variable `*pythagorean*`. For this store configuration the three variables `a`, `b`, `c` are restricted to the domain $\{1, \ldots, \mathtt{x}\}$ (Line 3). Furthermore, two additional constraints on these variables are defined. Finally, in Line 7 we initiate the full solving process by a search for all possible solutions. Note, that mathematical formulas may also be entered with the help of an infix-converting

[1] within the limits of technical and theoretical possibilities

```
(defun pythagorean-triples (x)
  (define-constraint-system *pythagorean* ()
    ((in (a b c) (1 . x))
     (<= a b)                ;avoid symmetric solutions
     (= (+ (^ a 2) (^ b 2))
        (^ c 2))))
  (search :find-all (a b c) *pythagorean*))
```

Figure 8.1.: The Pythagorean Triples problem as defined by CLFD's user interface language.

reader macro such that their appearance is in the more conventional notation such as `#I(a^2 + b^2 = c^2)`.

The clear and direct definition is possible because CLFD did fill in other parameters with default options. Figure 8.2 displays the complete definition which also shows the possible parameter customizations.

We particularly see that the propagation phase (Lines 3–5) and the search tree explorer (Lines 11–15) are customized by the different solver elements that were introduced in previous chapters. However, the user interface abstracts from the

```
1  (defun pythagorean-triples (x)
2    (define-constraint-system *pythagorean*
3        (:store copying-store
4         :domain integer-finite-domain
5         :scheduler basic-scheduler)
6      ((in (a b c) (1 . x))
7       (<= a b)
8       (= (+ (^ a 2) (^ b 2))
9          (^ c 2))))
10   (search :find-all (a b c) *pythagorean*
11           (:explore dfs
12            :branching (make-branching (a b c)
13                                       select-min-size-variable
14                                       select-min-value)
15            :restoration copying)))
```

Figure 8.2.: The Pythagorean Triples problem with explicit declaration of all default parameters.

low-level class instances that actually represent a constraint problem. The creation and parametrization of the required class instances is transparently handled by the interface layer.

The `search` function is used to initiate the solving process to find a solution to a defined constraint system. It takes at least three arguments: a goal, a list of variables and a constraint system. The result is a list of list where each list consists of the variable values of a particular solution in the order of the initially provide variable list. The solving process can be customized with additional parameters which influence the heuristic selection during search, the exploration strategy and the restoration approach.

Whereas the *stand-alone interface* is intended for users that are only interested in solving constraint systems, the *programmatic interface* is intended for programmers of applications, which employ constraint systems only as part of their overall operations. Hence we introduced a number of additional constructs that are specifically tailored to this setting, allowing the programmatic creation and solving of constraint systems, including the automatic processing of information contained in solutions to the constraint system.

The following macro is the most prominent feature in this regard. It makes use of a number of smaller programmatic interface functions we are not detailing in this overview.

with-solutions (((&rest *vars*) *store-form goal-form*) &body *body*) [*Macro*]

> Find the first solution of the solver obtained by evaluating *store-form*, which contains bindings for all the variables in *vars*. Once such a solution is found, the values the variables are bound to in the solution are extracted, and *body* is executed. In this execution environment the variables are bound to the extracted values. If no such solution is found, *body* is never executed. In either case the whole form always returns nil as its value. The search for further solutions continues until *goal-form* indicates that the goals is reached.

Instead of using the simple `search` function which simply returns a list of solution values, it is thus possible to evaluate complex application code whenever a solution is found. The respective variables are bound to their respective solution values at this point.

8.2. Managing Relation Operators

As can be seen in the example in Figure 8.1, the user does not need to reference actual propagator instances when describing constraint problems. The user interface

transparently abstracts from the propagator machinery and instead provides a description language that resembles the usual mathematical notation. This ambition results in a number of problems:

- Often, there are a number of propagators for a particular constraint, each providing the optimized handling of a specific case in addition to one generic propagator.

  One such example is the arithmetic equality expressions where CLFD provides optimized propagators for equations of the form $y = 2x$, $x = y + z$ and others (cf. Section 4.2) in addition to the automatic handling and splitting of general non-linear expressions. Remember, that propagators are defined on relations and not the arithmetic operations. Therefore, in this case we have to distinguish between the different kind of equality propagators.

- Since CLFD is designed to be an open and extensible system, it should be possible to add specializations of constraint propagators or new operators to the interface language.

In the next sections we illustrate our approach to solve and accomplish the above problems and criteria.

8.2.1. Adding New Constraint Relations

The interface layer uses relation operators to express the interconnections and dependencies of the individual propagators. The operator class is used to express and register a new operator to the system.

operator [*Standard Class*]

Abstract super-class of all relation operators as provided by the interface layer. The class has the following slots:

name contains the symbolic name of the defined operator.

equivalences contains a list of symbolic names of operators that entail an equivalent constraint relation or a number of constraint relations, which in conjunction express the constraint relation denoted by the symbol. For example the required additional constraints as listed in Table 5.2 can be defined in this way.

specializers contains a list of symbolic names of operators that provide (optimized) specializations of a generic operator as described by name.

To ease the definition of the operator hierarchy and required dependencies, we provide a macro that simplifies the definition by expanding into the actual class definitions and registering the individual relation operators to the interface. The resulting hierarchy is thus flexible and can be enhanced when new propagators are added to the solver.

define-operator (*name* (&rest *equivalences*) (&rest *specializers*)) [*Macro*]

Such relation operators represent the basic building blocks to express constraint systems at the user interface level. For example, the equivalence of *alldifferent* relations can thus be defined in the following way:

```
1 (define-operator alldifferent-instantiated () ())
2
3 (define-operator alldifferent-bounds () ())
4
5 (define-operator alldifferent
6     (alldifferent-instantiated alldifferent-bounds)
7     ())
```

The two relations are now defined to be equal. Using the `alldifferent` relation operator will result in the posting of the two equivalent relations to the solver store. The result is a cooperative pruning of the two propagation algorithms of both *all-different* propagators. This effect can also be used to post implied constraints for a relation, e.g. the ones listed in Table 5.2.

Relations that express particular specializations of a generic relation are defined as follows.

```
1 (define-operator =
2     ()
3     (=-shift =-ternary ....))
```

This code defines relation types of the form $x = y + \mathrm{c}$ and $x = y + z$ to be part of the generic equality hierarchy.

Now, we are able to express operator relations. However, it is not yet clear how actual propagator instances are posted to the constraint store. This is detailed in the next section.

8.2.2. Prioritizing, Matching and Grouping of Propagators

The relation hierarchy of the different operators as described in the previous section needs to be mapped into a respective number of propagator instances to resemble a particular constraint problem as defined by the user. There is a specific function initialize-constraint that is responsible for this part.

initialize-constraint (*operator store* &rest *operator-args*) [*Generic Function*]

> This generic function is specialized on *operator*. It is responsible for posting the *propagator* instance corresponding to an operator into *store*. This instance is initialized by using the arguments *operator-args*.

For *equivalences* this method is simply called for each equivalent operator. Thus, the result is a store with a number of constraint operators each expressing an equivalent relation. If this is not desired, then simply using an operator from a lower level of the equivalence hierarchy should be used directly (e.g. the `alldifferent-bounds` in the previous section).

For an operator that is subsuming the constraint relation of a number of specialized operators the situation is more complicated. In this case, initialize-constraint is responsible for selecting and posting the most specific propagator of a relation to the store. To simplify the description and selection this most specific operator, CLFD provides a matching language which can be used in addition to standard language constructs.

This pattern matching language is able to describe a constraint expression in Lisp s-expression form based on

- structure and nesting matching,
- type-based matching,
- (symbol name and numeric) equality-based matching.

When a match for a corresponding specific operator is found, its initialization method is called which then is responsible for posting the propagator instance to the store.

For example a matching of the constraint expression for the propagator representing $y = cx$ (where c is a constant integer) can be expressed in the following way by using the pattern mechanism:

```
1 (match (= ?var (* #'integerp ?var))
2        expression)
```

The above matcher is part of a longer selection process for polynomial expression. Specific instances are being tested for successful matching from most specific to most general. This is similar to constructor-based matching on function application in functional programming languages such as OPAL or HASKELL.

We provide an elaborate pattern language that is compiled into LISP code to gain efficient matchers without any run-time overhead due to interpretation.

9. Conclusion

We have developed a general framework of constraint solvers that is able to handle several different domains and accompanying pruning methods. The design is tailored to make use of and cushion the effects, characteristics and attributes of dynamic languages.

We started by giving an introduction to constraint solving and its theoretical model in Chapter 2. Additionally, in Section 2.1 we described the properties and advantages of dynamic languages.

The general design of a framework is a distinct and strict separation of the individual solver components: domain types, variables, propagators and the managing store in addition with a scheduler for the propagation phase. Based on the theoretical foundations as laid out in Sections 2.2 and 2.3 we derived the mathematical properties of the framework modules. Applying these attributes to the component connections we determined several protocols that enable a transparent module interaction while at the same time enforcing many of the fundamental properties.

Later, in Section 3.3, we discussed several implementation strategies for fundamental components. Especially the domain type of finite integer sets has been modeled with different data structures.

Furthermore, Chapter 4 gives an in-depth perspective to the propagation components: propagators and schedulers. Careful analysis provided the global properties and information for a well performing propagation phase. Several schedulers and attached propagator attributes which in part exploited the dynamic aspects of the host system where developed in response. Additionally, we implemented several complex propagators to test the framework. Starting from simple arithmetic constraints to sophisticated graph-based propagators.

An outstanding feature of Section 4.2.2 is the symbolic analysis, simplification and splitting of complex (arithmetic) constraint expressions into simpler, supported constraints.

Chapter 5 continues by deriving a highly abstracted search framework. Not only does it allow the easy development of complex search procedures by incrementally and locally refining existing search method implementations but it also enables the painless integration of problem specific and domain type dependent heuristics for variable selection and domain splitting.

By applying the mathematical module properties of the previous chapters we

are able to detail and to prove termination and confluence conditions in Chapter 6. Section 6.2 also illustrates and proves the connection of several employed optimization techniques to fix-point theory.

To evaluate the overall design decisions Chapter 7 compares the computation performance to selected related work of Section 2.5 by executing and analyzing a number benchmark problems. We begin by describing and advocating our statistical method in Section 7.1. Section 7.3 concludes our analysis and shows that our framework offers competitive performance compared to a highly sophisticated static solving system. Most of all, we display distinctly better performance to a previous dynamic approach.

Finally, the integration of constraints into the host language is presented in Chapter 8. Minimizing the impedance mismatch, complex constraint problems and solver specifics can be transparently integrated into the host system and host language.

The contributions of this thesis are threefold. First, we derived a highly flexible constraint framework that utilizes and cushions the dynamic nature of the underlying host system. A number of connection protocols for component connection and module exchange were defined and the necessary theoretical properties were outlined. These properties were used to argue about termination and confluence conditions.

Second, the model and several module realizations were implemented. Additionally, to demonstrate the flexibility and to experiment with different data structures, a number of implementation alternatives were prototyped.

Third, the experimental performance results were compared to alternative solver systems for several benchmark problems. We argue in support of our statistical approach and accomplish favorable performance numbers for our framework.

9.1. Future Work and Perspectives

We see several ways for future work and smaller improvements. First, there is always a need for more advanced and thus stronger propagators. In this respect it would be most advantageous to complete the work of the two diploma theses (cf. Section 4.5).

Furthermore, an extension of the symbolic simplifier could yield simplification on a wider range of operations. Currently, only integer-based polynomials are handled. An extension to interval (floating-point) based expressions is possible, including a canonical form for handling trigonometric operations.

From a theoretical perspective, a closer integration and analysis of the work on meta-solving and language integration as constraint solving [Hofstedt and Pepper, 2006] could be promising, especially with respect to a close integration within the framework and thus better solver performance than the translation burdened solver cooperation in [Hofstedt, 2001].

The current implementation could be evaluated for further internal optimization. Modules where the current dynamic flexibility is not necessary or hardly (if ever) exercised should be identified by carefully studying a wide rage of (additional) propagator implementations and their internal heuristics. For these areas the conversion to replace CLOS classes with standard structures (i.e. `defstruct`s) should reduce memory consumption (e.g. for problems with a huge number of variables) and should lead to better in-lining possibilities for the compiler.

A different approach would be the experimentation with an even more dynamic but problem specific function, quite similar to the run-time (optimized) code generation approach in [Grabmüller, 2009]. The framework could dynamically create a problem-specific propagator closure for each constraint in the CSP. This would eliminate run-time dispatch both for propagator instances as well as variable instances (but also makes class-redefinition at run-time as is currently possible, e.g. for size based domain re-encoding at run-time, a lot more difficult).

If one limits the number of possible propagator priority levels to a fixed set then a different scheduler queue in form of a radix heap [Cherkassky *et al.*, 1997] would be faster and more space efficient, especially when handling a great number of equally prioritized queue items.

9.1.1. Parallel Solving

The current trend in CPU design is a growing number of cores on a single system. To take advantage of this situation it would be advantageous to split the solving process into units that could be run in parallel.

An initial concept followed an approach that is similar to the design of software in the Erlang language [Amstrong, 2003]. In Erlang one creates a large number of independent threads that communicate and synchronize by means of asynchronous communication. We create a process for each node in the search tree and run a limited number in parallel (depending on the number of cores). This was supported by a parallel store that creates independent scheduler instances for each node and a parallel tree explorer. However, it turned out in our preliminary experiments that this design does not perform well since the propagation phase for each node is too small and the operating system scheduler of the Linux kernel was not distributing the processes to different cores. It would be interesting to explore other, more fruitful, approaches, e.g. by letting different process explore complete paths of the search tree.

A. Symbolic Simplification

In Example 4.2 of Section 4.2.2 we presented an already pre-simplified expression to illustrate the capabilities of the canonical form simplifier. For completeness we here display the original expanded expression in Lisp s-expression (or prefix) form. This example expression was presented by Bob Felts in the `comp.lang.lisp` news-group in a discussion on algebraic simplification (message id `1in6dgh.o04hgmch6wsgN%wrf3@stablecross.com`).

The simplifier is able to infer that the expression is equal to 0. Since this derivation only requires a few deterministic steps on the symbol level the simplified result is calculated in a fraction of a second.

This is in strong contrast to the involved data structures, splitting into sub-constraints for corresponding propagators, connection by additional auxiliary variables and backtracking search such as a pure constraint approach with actual domain determination would require for this result.

```
(/ (- (/ (* K (- K 1) (- K 2) (- K 3) (- K 4)) 120)
      (+ (* (/ (- (/ (* K (- K 1) (- K 2) (- K 3)) 24)
                  (+ (* (/ (- (/ (* K (- K 1) (- K 2)) 6)
                              (+ (* (/ (- (/ (* K (- K 1)) 2)
                                          (+ (* (/ (- K (* (/ (- 1 0) (+ K 1))
                                                           (/ (* (+ K 1) K) 2)))
                                                   (+ K 0))
                                                (/ (* (+ K -1) (+ K 0)) 2))
                                             (* (/ (- 1 0) (+ K 1))
                                                (/ (* (+ K -1) (+ K 0) (+ K 1))
                                                   6))))
                                       (+ K -1))
                                    (/ (* (+ K -2) (+ K -1)) 2))
                                 (* (/ (- K (* (/ (- 1 0) (+ K 1))
                                               (/ (* (+ K 1) K) 2)))
                                       (+ K 0))
                                    (/ (* (+ K -2) (+ K -1) (+ K 0)) 6))
                                 (* (/ (- 1 0) (+ K 1))
                                    (/ (* (+ K -2) (+ K -1) (+ K 0) (+ K 1))
                                       24))))
                           (+ K -2))
                        (/ (* (+ K -3) (+ K -2)) 2))
```

```
                (* (/ (- (/ (* K (- K 1)) 2)
                         (+ (* (/ (- K (* (/ (- 1 0) (+ K 1))
                                          (/ (* (+ K 1) K) 2)))
                                  (+ K 0))
                               (/ (* (+ K -1) (+ K 0)) 2))
                            (* (/ (- 1 0) (+ K 1))
                               (/ (* (+ K -1) (+ K 0) (+ K 1))
                                  6))))
                      (+ K -1))
                   (/ (* (+ K -3) (+ K -2) (+ K -1)) 6))
                (* (/ (- K (* (/ (- 1 0) (+ K 1))
                              (/ (* (+ K 1) K) 2))) (+ K
                                                       0))
                   (/ (* (+ K -3) (+ K -2) (+ K -1) (+ K 0)) 24))
                (* (/ (- 1 0) (+ K 1))
                   (/ (* (+ K -3) (+ K -2) (+ K -1) (+ K 0) (+ K 1))
                      120))))
             (+ K -3))
          (/ (* (+ K -4) (+ K -3)) 2))
       (* (/ (- (/ (* K (- K 1)) (- K 2)) 6)
             (+ (* (/ (- (/ (* K (- K 1)) 2)
                         (+ (* (/ (- K (* (/ (- 1 0) (+ K 1))
                                          (/ (* (+ K 1) K) 2)))
                                  (+ K 0))
                               (/ (* (+ K -1) (+ K 0)) 2))
                            (* (/ (- 1 0) (+ K 1))
                               (/ (* (+ K -1) (+ K 0) (+ K 1))
                                  6))))
                      (+ K -1))
                   (/ (* (+ K -2) (+ K -1)) 2))
                (* (/ (- K (* (/ (- 1 0) (+ K 1))
                              (/ (* (+ K 1) K) 2))) (+ K
                                                       0))
                   (/ (* (+ K -2) (+ K -1) (+ K 0)) 6))
                (* (/ (- 1 0) (+ K 1))
                   (/ (* (+ K -2) (+ K -1) (+ K 0) (+ K 1)) 24))))
             (+ K -2))
          (/ (* (+ K -4) (+ K -3) (+ K -2)) 6))
       (* (/ (- (/ (* K (- K 1)) 2)
                (+ (* (/ (- K (* (/ (- 1 0) (+ K 1))
                                 (/ (* (+ K 1) K) 2)))
                         (+ K 0))
                      (/ (* (+ K -1) (+ K 0)) 2))
                   (* (/ (- 1 0) (+ K 1))
                      (/ (* (+ K -1) (+ K 0) (+ K 1)) 6))))
             (+ K -1))
          (/ (* (+ K -4) (+ K -3) (+ K -2) (+ K -1)) 24))
       (* (/ (- K (* (/ (- 1 0) (+ K 1)) (/ (* (+ K 1) K) 2))) (+ K 0))
```

```
          (/ (* (+ K -4) (+ K -3) (+ K -2) (+ K -1) (+ K 0)) 120))
       (* (/ (- 1 0) (+ K 1))
          (/ (* (+ K -4) (+ K -3) (+ K -2) (+ K -1) (+ K 0) (+ K 1))
             720))))
  (+ K -4))
```

Example evaluation where the variable `*term*` is bound to the above expression:

```
CASI> (time (simplify *term*))
Evaluation took:
  0.001 seconds of real time
  0.000000 seconds of total run time (0.000000 user, 0.000000 system)
  0.00% CPU
  1,922,120 processor cycles
  24,544 bytes consed

⇒ 0
```

The `0` in the last line is the resulting simplified expression which is returned by the `simplify` function.

List of Figures

[1] The graphics are based on a map from the UK Office for National Statistics (`http://www.statistics.gov.uk/geography/downloads/uk_gor_cty.pdf`).

List of Algorithms

List of Tables

Bibliography

[Ahuja *et al.*, 1993] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993. 71

[Aït-Kaci and Nasr, 1989] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2(1):51–89, February 1989. 72

[Alt *et al.*, 1991] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a Maximum Cardinality Matching in a Bipartite Graph in Time $O(n^{1.5}\sqrt{m/\log n})$. *Inf. Process. Lett.*, 37(4):237–240, 1991. 70

[Amstrong, 2003] Joe Amstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Swedish Intitute of Computer Science (SICS), 2003. 131

[Anthony and Frisch, 1996] Simon Anthony and Alan M. Frisch. Towards Inductive Constraint Logic Programming. In *Working notes, 1996 AISB Workshop on Automated Reasoning: Bridging the Gap between Theory and Practice*, pages 3–4, 1996. 11

[Apt and Zoeteweij, 2004] Krzysztof R. Apt and Peter Zoeteweij. A Comparative Study of Arithmetic Constraints on Integer Intervals. In Krzysztof R. Apt, Francois Fages, Francesca Rossi, Peter Szeredi, and Jozsef Váncza, editors, *Proceedings of CSCLP 2003*, volume 3010 of *LNAI*, pages 1–24. Springer-Verlag, 2004. 61, 63, 64

[Apt and Zoeteweij, 2006] Krzysztof R. Apt and Peter Zoeteweij. An Analysis of Arithmetic Constraints on Integer Intervals. arXiv.org (cs.PL/0607016), 2006. 63, 67, 114, 116

[Apt, 2003] Krzysztof R. Apt. *Constraint Programming*. Cambridge University Press, 2003. 19, 21, 23, 54, 61

[Artiouchine *et al.*, 2005] Konstantin Artiouchine, Phillippe Baptiste, and Juliette Mattioli. On Modeling a Dynamic Hybrid System with Constraints: Computing

Aircraft Landing Trajectories. *INFORMS Journal on Computing*, February 2005. 50

[Azevedo, 2002] Francisco M. C. A. Azevedo. *Constraint Solving over Multi-valued Logics*. PhD thesis, Universidade Nova de Lisboa, 2002. 74

[Barnier and Brisset, 2001] Nicolas Barnier and Pascal Brisset. Facile: a Functional Constraint Library. In *Colloquium on Implementation of Constraint and LOgic Programming Systems 2001 (CLOPS'01)*, 2001. 26

[Beldiceanu *et al.*, 2005] N. Beldiceanu, M. Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog. Research Report T2005-08, Swedish Institute of Computer Science (SICS), 2005. 75

[Benhamou and Older, 1997] Frédéric Benhamou and William J. Older. Applying Interval Arithmetic to real, integer, and Boolean Constraints. *Journal of Logic Programming*, 32(1):1–24, July 1997. 115

[Bessière *et al.*, 2005] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005. 57

[Bobrow *et al.*, 1987] David G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. *X3J13 Document 87-002*, 1987. 36

[Bodirsky, 2004] Manuel Bodirsky. *Constraint Satisfaction with Infinite Domains*. PhD thesis, Humboldt-Universität Berlin, 2004. 19

[Bogen *et al.*, 1977] Richard Bogen, Jeffrey Golden, Michael Genesereth, and Alexander Doohovskoy. *MACSYMA reference manual*. Massachusetts Institute of Technology, Cambridge, MA, USA, 9 edition, 1977. 17

[Cai and Paige, 1989] Jiazhen Cai and Robert Paige. Program derivation by Fixed-Point Compuation. *Science of Computer Programming*, 11(3):197–261, April 1989. 95, 101, 102

[Carlsson, 2001] M. Carlsson. Finite Domain Constraints in SICStus Prolog. CICLOPS Workshop at CP'2001, invited talk, 2001. 25

[Cerwinski, 2007] Reiner Cerwinski. Regular-Language-Constraint für CLFD. unpublished, July 2007. 74

[Cheadle *et al.*, 2003] Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen, and Mark G. Wallace. ECLiPSe: An Introduction. Technical Report IC-Parc-03-1, Imperial College London, IC-PARC, 2003. 17, 25

[Cheney, 1970] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970. 90

[Cherkassky *et al.*, 1997] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. In *SODA '97: Proceedings of the eighth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 83–92, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics. 131

[Choi *et al.*, 2001] Chio Wo Choi, Martin Henz, and Ka Boon Ng. Components for State Restoration in Tree Search. In Toby Walsh, editor, *Principles and Practice of Constraint Programming – CP 2001*, volume 2239 of *Lecture Notes in Computer Sciece (LNCS)*. Springer-Verlag, 2001. 85, 86

[Dechter, 2003] Rina Dechter. *Constraint Programming*. Morgan Kaufmann Publishers, 2003. 57

[Dewdney. Alexander K., 1985] Dewdney. Alexander K. Computer Recreations. *Scientific American*, pages 16–26, December 1985. 115

[Dincbas *et al.*, 1988] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, and Alexander Herold. The CHIP System: Constraint Handling in Prolog. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 774–775, London, UK, 1988. Springer-Verlag. 11, 25

[Dooms *et al.*, 2005] G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In Peter van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *Lecxture Notes in Computer Sciences (LNCS)*. Springer-Verlag, 2005. 19

[Dubrovsky and Shvetsov, 1995] V. Dubrovsky and A. Shvetsov. Number system unknown (B144). *Quantum CyberTeaser: May/June*, 1995. 114

[Ehrig *et al.*, 2001] Hartmut Ehrig, Bernd Mahr, Felix Cornelius, Martin Große-Rhode, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag, 2 edition, 2001. 20

[Erwig, 1998] Martin Erwig. Diets for Fat Sets. *Journal of Functional Programming*, 8(6):627–632, 1998. 51

[Fahrmeir *et al.*, 2007] Ludwig Fahrmeir, Rita Künstler, Iris Pigeot, and Gerhard Tutz. *Statistik: Der Weg zur Datenanalyse*. Springer-Verlag, 2007. 110

[Frank and Hofstedt, 2005] Stephan Frank and Petra Hofstedt. CLFD: A Finite Domain Constraint Solver in Common Lisp. In Carl Shapiro and JonL White, editors, *Proceedings of the 25th International Lisp Conference (ILC 2005)*, pages 159–168, Stanford University, Palo Alto, July 2005. Association of Lisp Users (ALU), Franz Inc. 33, 121

[Frank and Mai, 2002] Stephan Frank and Pierre R. Mai. Strategies for Cooperating Constraint Solvers. Diploma thesis, TU Berlin, July 2002. 79

[Frank *et al.*, 2003a] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. A flexible Meta-Solver Framework for Constraint Solver Collaboration. In A. Günter, R. Kruse, and B. Neumann, editors, *Proceedings of the 26th German Conference on Artificial Intelligence - KI 2003*, volume 2821 of *LNCS*. Springer-Verlag, 2003. 29, 79

[Frank *et al.*, 2003b] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. Meta-S: A Strategy-oriented Meta-Solver Framework. In Ingrid Russell and Susan Haller, editors, *Proceedings of the 16th International Florida Artificial Intelligence Research Symposium Conference (FLAIRS)*. The AAAI Press, May 2003. 27

[Frank *et al.*, 2004] Stephan Frank, Petra Hofstedt, and Dirk Reckmann. Strategies for the Efficient Solution of Hybrid Constraint Logic Programs. In S. Muñoz-Hernández, J. M. Gómez-Perez, and P. Hofstedt, editors, *WLPE2004: 14th Workshop on Logic Programming Environments and MultiCPL2004: Third Workshop on Multiparadigm Constraint Programming Languages. Workshop Proceedings*, 2004. 27

[Frank *et al.*, 2005] Stephan Frank, Petra Hofstedt, and Dirk Reckmann. System Description: Meta-S – Combining Solver Cooperation and Programming Languages. In *W(C)LP 2005: 19th Workshop on (Constraint) Logic Programming. Workshop Proceedings*, Technical Report, University of Ulm, 2005. 29, 79

[Frank, 2007] Stephan Frank. Constraint Solving in Common Lisp. In JonL White, editor, *26th International Lisp Conference (ILC 2007)*, pages 45–54, Clare College, Cambridge, United Kingdom, April 2007. Association of Lisp Users, ACM Press. 79, 82

[Fredman *et al.*, 1986] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica*, 1(1):111–129, 1986. 59

[Garns, 1976] Howard Garns. Number Place. Dell Pencil Puzzles & Word Games, Issue #16, May 1976. 111

[Gennari, 2002] Rosella Gennari. *Mapping Inferences: Constraint Propagation and Diamond Satisfaction*. PhD thesis, Universiteit van Amsterdam, December 2002. 95, 101

[Gent and Walsh, 1999] Ian P. Gent and Toby Walsh. CSPlib: a benchmark library for constraints. In Joxan Jaffar, editor, *Proceedings of Principles and Practice of Constraint Programming – CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999*, volume 1713 of *Lecture Notes in Computer Science (LNCS)*, pages 480–481, London, UK, October 1999. Springer-Verlag. 115

[Georges *et al.*, 2007] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Oopsla'07: Proceedings of the 22nd annual ACM Sigplan Conference on Object Oriented Programming Systems and Applications*, pages 57–76, New York, NY, USA, 2007. Association of Computing Machinery (ACM), ACM Press. 107, 108

[Grabmüller, 2009] Martin Grabmüller. *Dynamic Compilation of Functional Programs*. PhD thesis, Technical University Berlin, 2009. 131

[Hanus *et al.*, 1995] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995. 11

[Harvey and Ginsberg, 1995] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 607–615, Montréal, Québec, Canada, August 1995. Morgan Kaufmann. 89

[Hein, 2002] James L. Hein. *Discrete Structures, Logic, and Computability*. Jones and Bartlett Computer Science, second edition edition, 2002. 95

[Henz *et al.*, 1999] M. Henz, T. Müller, and K. B. Ng. Figaro: Yet Another Constraint Programming Library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, 1999. 26, 33

[Hickey *et al.*, 2001] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval Arithmetic: from Principles to Implementation. *Journal of the ACM*, 5(48):1038–1068, September 2001. 19, 92, 93

[Hoche *et al.*, 2003] M. Hoche, H. Müller, H. Schlenker, and A. Wolf. FirstCS - A Pure Java Constraint Programming Engine. In *Proceedings of the 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'03*, January 2003. 26

[Hofstedt and Pepper, 2006] Petra Hofstedt and Peter Pepper. Integration of Declarative and Constraint Programming. *Theory and Practice of Logic Programming (TPLP): Special Issue on Multiparadigm Languages and Constraint Programming*, 2006. 27, 131

[Hofstedt, 2001] Petra Hofstedt. *Cooperation and Coordination of Constraint Solvers*. PhD thesis, Technische Universität Dresden, March 2001. Shaker Verlag, Aachen. 18, 27, 32, 131

[Hofstedt, 2010] Petra Hofstedt. Constraint-Based Object-Oriented Programming. *IEEE Software*, 27:53–56, 2010. 121

[Hölzl, 2001] Matthias M. Hölzl. *Constraint-Lambda Calculi – Theory and Applications*. PhD thesis, Ludwig-Maximilians Universität München, 2001. 79

[Hölzl, 2002] M. Hölzl. ConS/Lisp – A Metaobject-Protocol based Nondeterministic Lisp. In *Proceedings of the International Lisp Conference (ILC)*, 2002. 79

[Hwang and Mitchell, 2005] Joey Hwang and David G. Mitchell. 2-Way vs. *d*-Way Branching for CSP. In Peter van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005*, number 3709 in LNCS, pages 343–357. Springer-Verlag, 2005. 87

[IEEE, 1987] IEEE. IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, 1987. Reprinted in SIGPLAN 22, 2, 9-25. 92

[Jones *et al.*, 1993] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, and Phil Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference (93)*, 1993. 16

[Jones, 1999] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Memory Management*. John Wiley & Sons Ltd., 1999. 16, 90

[Keene, 1989] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley/Symbolic Press, 1989. 36

[Kiczales and Rivieres, 1991] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. 13, 17, 36

[Kirschke, 1997] Heiko Kirschke. *Persistenz in objekt-orientierten Programmiersprachen*. PhD thesis, Humboldt Universität Berlin, 1997. 17

[Kleene, 1952] Stephen Cole Kleene. *Introduction to Metamathematics*. Bibliotheca Matematica. North-Holland Publishing Company, Amsterdam, 1952. 97

[Koa, 2004] Koalog S.A.R.L., `www.koalog.com`. *Koalog: An Overview of Koalog Constraint Solver*, 2004. 26

[Laburthe, 2000] François Laburthe. CHOCO: implementing a CP kernel. In *Techniques foR Implementing Constraint Programming Systems (TRICS workshop)*, September 2000. 27

[Land and Doig, 1960] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960. 88

[Lilja, 2000] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, New York, NY, USA, August 2000. 108, 110

[Lopez-Ortiz *et al.*, 2003] Alejandro Lopez-Ortiz, Claude-Guy Qimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*. Acapulco, Mexico, Morgan Kaufmann, August 2003. 69

[MacLachlan, 1992] Robert A. MacLachlan. The Python Compiler for CMU Common Lisp. In *Lisp and Functional Programming*, pages 235–246, 1992. 15

[MacLachlan, 1994] Robert MacLachlan. Design of CMU Common Lisp. Technical Report CMU-CS-91-108, Carnegie Mellon University (CMU), 1994. 16

[Marriott and Stuckey, 1998] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998. 19

[Martin and Moses, 1970] W. A. Martin and J. Moses. Mathlab (MACSYMA). MAC Programming Report V, Massachusetts Institute of Technology, A. I. Lab., Cambridge, MA, USA, 1970. 65

[Meijer and Drayton, 2004] Erik Meijer and Peter Drayton. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. In Wolfgang de Meuter, editor, *International Workshop on Revival of Dynamic Languages at OOPSLA'04*, 2004. 15

[Müller and Müller, 1997] Tobias Müller and Martin Müller. Finite Set Constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *Proceedings of the 13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997. 33

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. On the Revival of Dynamic Languages. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition, 4th International Workshop, SC 2005, Edinburgh, UK, April 9, 2005, in conjunction with OOPSLA'04 (revised selected papers)*, volume 3628 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2005. 16

[Niesche, 2007] Harald Niesche. Set-Domain Repräsentation und Set-Constraints für CLFD. unpublished, May 2007. 74

[Norvig, 1992] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, 1992. 65

[Norvig, 1998] Peter Norvig. Design Patterns in Dynamic Programming. Tutorial at Object World, 1998. 16

[Norvig, 2006] Peter Norvig. Solving Every Sudoku Puzzle. Online Essay (`http://norvig.com/sudoku.html`, last visited 2008-09-19), June 2006. 111

[Nuutila and Soisalon-Soininen, 1994] Esko Nuutila and Eljas Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. *Information Processing Letters*, 49(1):9–14, 1994. 70

[Paulson, 2007] Linda Dailey Paulson. Developers Shift to Dynamic Programming Languages. *IEEE Computer*, 40(2):12–15, February 2007. 15

[Pepper and Hofstedt, 2006] Peter Pepper and Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmiertechnik*. Springer-Verlag, 2006. 95, 102, 103

[Pepper, 1991] Peter Pepper. The Programming Language OPAL. Technical report, Technische Universität Berlin, 91-10, 1991. 16

[Pesant, 2004] Gilles Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *LNCS*. Springer-Verlag, 2004. 74

[Puget, 1992] Jean-François Puget. PECOS: a high level constraint programming language. In *Proceedings of First Singapore International Conference on Intelligent Systems (SPICIS'92)*, 1992. 26

[Puget, 1994] J.-F. Puget. A C++ Implementation of CLP. In *Proceedings of the Singapore International Conference on Intelligent Systems (SPICIS)*, Singapore, November 1994. 12, 26

[Quimper *et al.*, 2004] Claude-Guy Quimper, Alejandro Lopez-Ortiz, Peter van Beek, and Alexander Golynski. Improved algorithms for the global cardinality constraint. In M. Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP2004)*, volume 3258 of *Lecture Notes in Computer Science (LNCS)*. Toronto, Canada, Springer-Verlag, September 2004. 71

[Régin, 1994] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI-94*, 1994. 70

[Régin, 2003] Jean-Charles Régin. Global Constraints and Filtering Algorithms. In Michela Milano, editor, *Constraint and Integer Programming: Toward a Unified Methodology*, number 27 in Operations Research/Computer Science Interfaces, chapter 4, pages 89–129. Kluwer Academic Publishers, 2003. 114

[Rigo, 2004] Armin Rigo. Representation-based just-in-time Specialization and the Psyco Prototype for Python. In *Pepm '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 15–26, New York, NY, USA, 2004. ACM. 15

[Ringwelski and Hoche, 2005] Georg Ringwelski and Matthias Hoche. Impact- and Cost-Oriented Propagator Scheduling for Faster Constraint Propagation. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *19th Workshop on (Constraint) Logic Programming (W(C)LP)*, volume 2005-1 of *Ulmer Informatik-Berichte*. Universität Ulm, Germany, February 2005. 58

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006. 25

[Schmid and Waltermann, 2004] Ute Schmid and J. Waltermann. Automatic Synthesis of XSL-Transformations from Example Documents. In M.H. Hamza, editor, *Artificial Intelligence and Applications Proceedings (AIA 2004)*. Acta Press, 2004. 11

[Schmid *et al.*, 2002] Ute Schmid, Marina Müller, and Fritz Wysotzki. Integrating Function Application in State-Based Planning. In Matthias Jarke, Jana Koehler, and Gerhard Lakemeye, editors, *KI 2002: Advances in Artificial Intelligence, 25th Annual German Conference on AI*, volume 2479 of *Lecture Notes in Computer Science*, pages 144–162. Springer-Verlag, 2002. 11

[Schulte and Carlsson, 2006] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, Amsterdam, The Netherlands, 2006. 26, 47

[Schulte and Stuckey, 2004] Christian Schulte and Peter J. Stuckey. Speeding Up Constraint Propagation. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag. 58

[Schulte and Stuckey, 2008] Christian Schulte and Peter J. Stuckey. Efficient Constraint Propagation Engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008. 26

[Schulte, 1999] Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, November 1999. The MIT Press. 85

[Siek and Vachharajani, 2008] Jeremy G. Siek and Manish Vachharajani. Gradual Typing with Unification-based Inference. In *DLS '08: Proceedings of the 2008 Symposium on Dynamic Languages*, pages 1–12, New York, NY, USA, 2008. ACM Press. 16

[Siskind and McAllester, 1993a] J. M. Siskind and D. A. McAllester. Nondeterministic Lisp as a Substrate for Constraint Logic Programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*. AAAI Press, 1993. 27, 114

[Siskind and McAllester, 1993b] J. M. Siskind and D. A. McAllester. Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp. Technical Report IRCS–93–03, ICRS, 1993. 12, 27, 29

[Sleator and Tarjan, 1985] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3), 1985. 51

[Smith *et al.*, 1999] Barbara Smith, Kostas Stergiou, and Toby Walsh. Modelling the Golomb Ruler Problem. Technical Report APES-13-1999, APES Research Group, June 1999. 115

[Smith, 2005] Babera M. Smith. Modelling for Constraint Programming. Tutorial at the 1st International CP Summer School, 2005. 50

[Sussman, 1975] Gerald Jay Sussman. Scheme: An Interpreter for Extended Lambda Calculus. In *Memo 349, MIT AI Lab*, 1975. 27

[Sutherland, 1963] I. Sutherland. Sketchpad: A man-machine graphical comunication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963. 25

[Tack, 2009] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. Doctoral Dissertation, Saarland University, 2009. 26

[Tamura *et al.*, 2009] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling Finite Linear CSP into SAT. *Constraints*, 14(2):254–272, 2009. 27

[Tarski, 1955] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. 99

[Van Hentenryck and Michel, 2005] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005. 121

[Van Hentenryck, 1999] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, USA, 1999. 26

[Wallace *et al.*, 1997] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, August 1997. 25

[Wallace, 1993] Richard J. Wallace. Why AC-3 is Almost Always Better Than AC4 for Establishing Arc Consistency in CSPs. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 239–247. Chambéry, France, Morgan Kaufmann, September 1993. 55, 57

[White and Sleeman, 1998] S. White and D. Sleeman. Constraint Handling in Common Lisp. Technical report, University of Aberdeen, King's College, Scotland, U.K., December 1998. 27, 111

[White, 2000] Simon White. *Enhancing Knowledge Acquisition with Constraint Technology*. PhD thesis, University of Aberdeen, 2000. 27, 111

[Wolfram, 1991] Stephan Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 2 edition, 1991. 17

[Yermolovich *et al.*, 2009] Alexander Yermolovich, Christian Wimmer, and Michael Franz. Optimization of Dynamic Languages using Hierarchical Layering of Virtual Machines. In *DLS '09: Proceedings of the 5th Symposium on Dynamic Languages*, pages 79–88, New York, NY, USA, 2009. ACM. 15

[Zoeteweij, 2005] Peter Zoeteweij. *Composing Constraint Solvers*. PhD thesis, Universiteit van Amsterdam, November 2005. 12, 23, 27, 63, 78, 112

Index