Contents

Preface *page* 1

Part I CHR Tutorial 9

1 Getting Started 12
1.1 How CHR Works 12
1.1.1 Propositional Rules 12
1.1.2 Logical Variables 15
1.1.3 Built-In Constraints 16
1.2 CHR Programs and their Execution 18
1.2.1 Concrete Syntax 18
1.2.2 Informal Semantics 20
1.3 Exercises 21
1.4 Origins and Applications of CHR 24

2 My First CHR Programs 26
2.1 CHR as a Database Language 26
2.2 Multiset Transformation 28
2.2.1 Minimum 28
2.2.2 Boolean Exclusive Or 31
2.2.3 Greatest Common Divisor 32
2.2.4 Prime Numbers Sieve of Eratosthenes 34
2.2.5 Exchange Sort 35
2.2.6 Newton's Method for Square Roots 36
2.3 Procedural Algorithms 37
2.3.1 Maximum 37
2.3.2 Fibonacci 37
2.3.3 Depth First Search in Trees 40
2.3.4 Destructive Assignment 41

2.4 Graph-Based Algorithms 42
2.4.1 Transitive Closure 42
2.4.2 Partial Order Constraint 45
2.4.3 Grammar Parsing 46
2.4.4 Ordered Merging and Sorting 47
2.5 Exercises 50

Part II The CHR Language 57

3 Syntax and Semantics 60
3.1 Preliminaries 60
3.1.1 Syntactic Expressions 60
3.1.2 Substitution, Variants and Equality 61
3.1.3 Constraint Systems 62
3.1.4 Transition Systems 62
3.2 Abstract Syntax 64
3.3 Operational Semantics 66
3.3.1 Very Abstract Semantics 66
3.3.2 CHR with Disjunction - CHR$^\vee$ 68
3.3.3 Abstract Semantics ω_t 70
3.3.4 Refined Operational Semantics ω_r 75
3.4 Declarative Semantics 80
3.4.1 First-Order Logic Declarative Semantics 80
3.4.2 Linear Logic Declarative Semantics 86
3.5 Bibliographic Remarks 92

4 Properties of CHR 94
4.1 Anytime Approximation Algorithm Property 94
4.2 Monotonicity and Online Algorithm Property 96
4.3 Declarative Concurrency and Logical Parallelism 97
4.4 Computational Power and Expressiveness 103
4.5 Bibliographic Remarks 105

5 Program Analysis 107
5.1 Termination 107
5.1.1 Rankings 108
5.1.2 Program Termination 110
5.1.3 Derivation Lengths 111
5.2 Confluence 112
5.2.1 Minimal States 112
5.2.2 Joinability 113
5.2.3 Confluence Test 113

5.2.4 Joinability for Confluence 115
5.2.5 Examples 116
5.2.6 Confluence Test for the Abstract Semantics 119
5.2.7 Properties of Confluent Programs 121
5.3 Completion 123
5.3.1 Completion Algorithm 123
5.3.2 Examples 125
5.3.3 Correctness 128
5.3.4 Failing Completion and Inconsistency 129
5.3.5 Program Specialization by Completion 130
5.4 Modularity of Termination and Confluence 133
5.4.1 Modularity of Termination 134
5.4.2 Modularity of Confluence 138
5.5 Operational Equivalence 139
5.5.1 Operational Equivalence of Programs 139
5.5.2 Operational Equivalence of Constraints 140
5.5.3 Removal of Redundant Rules 143
5.6 Worst-Case Time Complexity 146
5.6.1 Simplification Rules 147
5.6.2 Programs 148
5.7 Bibliographic Remarks 150

6 Rule-Based and Graph-Based Formalisms in CHR 152
6.1 Rule-Based Systems 155
6.1.1 Production Rule Systems 156
6.1.2 Negation-as-Absence 158
6.1.3 Conflict Resolution 162
6.1.4 Event-Condition-Action Rules 163
6.1.5 Logical Algorithms Formalism 165
6.2 Rewriting-Based and Graph-Based Formalisms 167
6.2.1 Term Rewriting Systems 168
6.2.2 Multiset Transformation 173
6.2.3 Petri Nets 173
6.3 Constraint-Based and Logic-Based Programming 175
6.3.1 Prolog and Constraint Logic Programming 176
6.3.2 Concurrent Constraint Programming 178
6.4 Bibliographic Remarks 180

Part III CHR Programs and Applications 183

7 My First CHR Programs, Revisited for Analysis 186
7.1 Multiset Transformation 186
7.1.1 Minimum 186
7.1.2 Boolean Exclusive Or 190
7.1.3 Greatest Common Divisor 191
7.1.4 Primes Sieve of Eratosthenes 194
7.1.5 Exchange Sort 195
7.1.6 Newton's Method for Square Roots 198
7.2 Procedural Algorithms 198
7.2.1 Maximum 198
7.2.2 Fibonacci Numbers 199
7.2.3 Depth First Search in Trees 202
7.2.4 Destructive Assignment 203
7.3 Graph-Based Algorithms 203
7.3.1 Transitive Closure 203
7.3.2 Partial Order Constraint 206
7.3.3 Grammar Parsing 207
7.3.4 Ordered Merging and Sorting 208
7.4 Exercises 210

8 Finite Domain Constraint Solvers 213
8.1 Booleans $\mathcal{B}$ 214
8.1.1 Local Propagation 214
8.1.2 Boolean Cardinality 216
8.1.3 Clauses and Resolution 218
8.2 Path and Arc Consistency 220
8.2.1 Constraint Networks and Operations 221
8.2.2 Path Consistency 222
8.2.3 Finite Domain Arc Consistency 224
8.2.4 Temporal Reasoning with Path Consistency 227
8.3 Exercises 229
8.4 Bibliographic Remarks 233

9 Infinite Domain Constraint Solvers 234
9.1 Linear Polynomial Equation Solving $\Re$ 234
9.1.1 Variable Elimination 234
9.1.2 Gaussian-Style Elimination 235
9.1.3 Fourier's Algorithm 237
9.2 Lexicographic Order Global Constraint 239
9.2.1 Step-Wise Implementation 239
9.2.2 Constraint Solver 242

9.2.3 Worst-Case Time Complexity 243
9.2.4 Confluence 244
9.2.5 Logical Correctness 245
9.2.6 Completeness 246
9.3 Description Logic DL 248
9.3.1 Syntax and Semantics 248
9.3.2 Constraint Solver 249
9.3.3 Extensions 253
9.3.4 DL with Rules 253
9.4 Rational Trees RT 255
9.4.1 Constraint Solver 256
9.5 Feature Terms FT 261
9.6 Exercises 263
9.7 Bibliographic Remarks 267

10 Union-Find Algorithm 268
10.1 Union-Find Algorithm (UF) 269
10.1.1 Basic Union-Find 269
10.1.2 Optimized Union-Find 274
10.1.3 Complexity 276
10.2 Rational Tree Unification with Union-Find 277
10.3 Parallelizing Union-Find (PUF) 278
10.3.1 Basic Union-Find 279
10.3.2 Optimized Union-Find 280
10.3.3 Correctness and Complexity 282
10.4 Generalizing Union-Find (GUF) 283
10.4.1 Generalized Union-Find 283
10.4.2 Boolean Equations 289
10.4.3 Linear Polynomials 290
10.5 Bibliographic Remarks 291

Bibliography 293
[Press Text / Back Cover] 304
List of Figures 307
List of Illustrations 307
Index 308

Preface

The more constraints one imposes, the more one frees one's self.

Igor Stravinsky

CHR has taken off. After five dedicated workshops, two special journal issues and hundreds of related research articles, it was time to write this book about CHR.

About this book

This book is about programming with rules. It presents a rule-based constraint programming language called CHR (short for Constraint Handling Rules). While conceptually simple, CHR embeds the essential aspects of many rule-based and logic-based formalisms and can implement algorithms in a declarative yet highly effective way. The combination of information propagation and multiset transformation of relations in a concurrent, constraint-based language makes CHR a powerful declarative tool for knowledge representation and reasoning. Over the last decade CHR has not only cut its niche as a special-purpose language for writing constraint solvers, but has matured into a general-purpose language for computational logic and beyond.

This intermediate level book with a gentle introduction and more advanced chapters gives an overview of CHR for readers of various levels of experience. The book is addressed to researchers, lecturers, graduate students and professional programmers interested in languages for innovative applications. The book supports both self-study and teaching. It is accompanied by a website at `chr.informatik.uni-ulm.de`.

In short, this book concentrates on the basics of CHR while keeping in mind dozens of research papers. In 2009, there is a companion book on

recent advances in CHR and a survey article in a journal. A book on implementation of CHR and a collection of classical CHR papers is also planned.

Underlying Concepts

CHR relies on three essential concepts: rules, declarativity and constraints.

Rules are common in everyday life. The formalization of these rules goes back more than 2000 years to the syllogisms of the Greek philosopher Aristotle, who invented logic this way. Nowadays, rule-based formalisms are ubiquitous in computer science, from theory to practice, from modeling to implementation, from inference rules and transition rules to business rules.

Rules have a double nature, they can express monotonic static causal relations on the basis of logic, but also nonmonotonic dynamic behavior by describing state changes. Executable rules are used in declarative programming languages, in program transformation and analysis, and for reasoning in artificial intelligence applications. Such rules consist of a data description (pattern) and a replacement statement for data matching that description. Rule applications cause transformations of components of a shared data structure (e.g., constraint store, term, graph, or database).

Matured rule-based programming experiences a renaissance due to its applications in areas such as business rules, semantic web, computational biology, medical diagnosis, software verification, and security. Commonplace uses of rules are in insurance and banking applications, for mail filtering and product configuration.

Declarativity means to describe knowledge about entities, their relationships and states, and to draw inferences from it to achieve a certain goal, as opposed to procedural or imperative programs that give a sequence of commands to compute a certain result. Declarative program constructs are often related to an underlying formal logic.

Declarativity facilitates program development (specification, implementation, transformation, combination, maintenance) and reasoning about programs (e.g. correctness, termination, complexity). Declarative programming also offers solutions to interaction, communication, distribution and concurrency of programs.

Constraint reasoning allows one to solve problems by simply stating constraints (conditions, properties) which must be satisfied by a solution of the problem. A special program (the constraint solver) stores, combines, and simplifies the constraints until a solution is found. The partial solutions can be used to influence the run of the program that generates the constraints.

Programming by asserting constraints makes it possible to model and

specify problems with uncertain or incomplete information and to solve combinatorial problems, such as scheduling and planning. The advantages of constraint-based programming are declarative problem modeling on a solid mathematical basis and propagation of the effects of decisions expressed as additional constraints. The conceptual simplicity and efficiency of constraint reasoning leads to executable specifications, rapid prototyping, and ease of maintenance.

Constraint Handling Rules (CHR)

CHR is both a theoretical *formalism* (like term rewriting and Petri nets) related to a subset of first-order logic and linear logic, and a practical programming language (like Prolog and Haskell). CHR tries to bridge the gap between theory and practice, between logical specification and executable program by abstraction through constraints and the concepts of computational logic. By the notion of constraint, CHR does not distinguish between data and operations, rules are both descriptive and executable.

CHR is a declarative concurrent committed-choice constraint logic *programming language* consisting of guarded rules that transform multisets of constraints (relations, predicates). CHR was motivated by the inference rules that are traditionally used in computer science to define logical relationships and arbitrary fixpoint computations in the most abstract way.

Direct *ancestors of CHR* are logic programming, constraint logic programming, and concurrent committed-choice logic programming languages. Like these languages, CHR has an operational semantics describing the execution of a program and a declarative semantics providing a logical reading of the program which are closely related. Other influences were multiset transformation systems, term rewriting systems, and, of course, production rule systems. CHR embeds essential aspects of these and other rule-based systems such as constraint programming, graph transformation, deductive databases, and Petri nets, too.

In CHR, one distinguishes two main kinds of rules: *Simplification rules* replace constraints by simpler constraints while preserving logical equivalence, e.g., $\texttt{X}\leq\texttt{Y}\wedge\texttt{Y}\leq\texttt{X} \Leftrightarrow \texttt{X}=\texttt{Y}$. *Propagation rules* add new constraints that are logically redundant but may cause further simplification, e.g., $\texttt{X}\leq\texttt{Y}\wedge\texttt{Y}\leq\texttt{Z} \Rightarrow \texttt{X}\leq\texttt{Z}$. Together with $\texttt{X}\leq\texttt{X} \Leftrightarrow \texttt{true}$, these rules encode the axioms of a partial order relation. The rules compute its transitive closure and replace $\leq$ by equality ($=$) whenever possible.

Multi-headed rules allow to express complex interactions in a compact way. They provide for implicit iteration instead of cumbersome looping

constructs. In other words, CHR supports a topological view of structured data. Components can be directly accessed by just mentioning them in the rule head. CHR also allows for recursive descent where one walks through data.

CHR is appealing for applications in *computational logic*: Logical theories are usually specified by implications and logical equivalences that correspond to propagation and simplification rules. On the meta-level, given the transformation rules for deduction in a calculus, its inference rules map to propagation rules and replacement rules to simplification rules. In this context, CHR integrates deduction and abduction, bottom-up and top-down execution, forward and backward chaining, tabulation and integrity constraints.

Algorithms are often specified using inference rules, rewrite rules, sequents, proof rules, or logical axioms that can be directly written in CHR. Starting from such an executable specification, the rules can then be refined and adapted to the specifics of the application. Yet, CHR is no theorem prover, but an efficient programming language: CHR uses formulas to derive new information, but only in a restricted syntax (e.g., no negation) and in a directional way (e.g., no contrapositives) that makes the difference between the art of proof search and an efficient programming language.

The use of CHR as a *general-purpose programming language* is justified by the following observation: Given a state transition system, its transition rules can readily be expressed with simplification rules. In this way, CHR accounts for the double nature (causality versus change) of rules. *Statefulness* and declarativity are reconciled in the CHR language. Dynamics and changes (e.g., updates) can be modeled, possibly triggered by events and handled by actions (that can all be represented by constraints). CHR allows for explicit state (constraints, too), so that the efficiency of imperative programs can be achieved.

CHR programs have a number of *desirable properties* guaranteed and can be analyzed for others. Every algorithm can be implemented in CHR with best known time and space complexity, something that is not known to be possible in other pure declarative programming languages. The efficiency of the language is empirically demonstrated by recent optimizing CHR compilers that compete well with both academic and commercial rule-based systems and even classical programming languages.

Any CHR program will by nature implement an anytime (approximation) and online (incremental) algorithm. Confluence of rule applications and operational equivalence of programs are decidable for terminating CHR programs. We do not know of any other programming language in practical

use where operational equivalence is decidable. CHR does not have bias towards sequential implementation. A terminating and confluent CHR program can be run in parallel without any modification and without harming correctness. This property is called declarative concurrency (logical parallelism).

CHR does not necessarily impose itself as a new programming language, but as a language extension that blends in with the syntax of its *host language*, be it Prolog, Lisp, Haskell, C or Java. In the host language, CHR constraints can be posted and inspected; in the CHR rules, host language statements can be included.

CHR has been used for such *diverse applications* as type system design for Haskell, time tabling for universities, optimal sender placement, computational linguistics, spatio-temporal reasoning, chip card verification, semantic web information integration, computational biology, and decision support for cancer diagnosis. Commercial application include stock broking, optical network design, injection mould design and test data generation.

If asked what distinguishes CHR from similar programming languages and formalisms, the quick answer is that CHR is both a theoretical formalism and a practical programming language. CHR is the synthesis of multiset transformation, propagation rules, logical variables and built-in constraints into one conceptually simple language with a foundation in logic and with formal methods for powerful program analysis.

In summary, CHR is the "best computational formalism *and* the best programming language you have never seen in your life".

Contents

This book has three parts. The first part is a tutorial on CHR. The second part formally defines syntax and semantics of CHR, its properties and their analysis. The third part presents CHR programs and applications to which the analysis of Part II is applied. We present exercises and selected solutions for the chapters that contain practical programs in Part I and Part III of this book.

In **Part I**, the CHR tutorial tells you how to write CHR programs in one of the recent CHR libraries, how CHR rules look like and how rules are executed. A wealth of small but expressive example programs, often consisting of just one rule, are discussed in detail. The behavior of CHR implementations is explained, and different programming styles are exhibited: CHR as database language, for multiset transformation, for procedural algorithms

and for constraint solving. A special emphasis is placed on graph-based algorithms. The properties of the programs are discussed informally, and this foreshadows their thorough analysis in Part II of the book.

In **Part II**, syntax and semantics of CHR are formally introduced. We distinguish between a declarative semantics that is based on a logical reading of the rules and a operational semantics that describes how rules are applied. Several widely used variants of both types of semantics are given.

In the next chapter, guaranteed properties of CHR are discussed: The anytime algorithm property means that we can interrupt the program at any time and restart from the intermediate result. The online algorithm property means that we can add additional constraints incrementally, while the program is running. We then discuss declarative concurrency (also called logical parallelism). Last but not least, we show that CHR can implement any algorithm with best known time and space complexity.

The chapter on program analysis discusses termination, confluence, operational equivalence and time complexity: Since CHR is Turing-complete, termination is undecidable. Confluence of a program guarantees that a computation has the same result no matter which of the applicable rules are applied. Confluence for terminating CHR programs is decidable. Nonconfluent programs can be made confluent by completion which is introduced next. Modularity of termination and confluence under union of programs is discussed. Then, we give a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent programs. Finally, a meta-theorem to predict the worst-case time complexity of a class of CHR programs is given.

In the last chapter of this part, CHR is compared to other formalisms and languages by embedding them in CHR. It is shown that essential aspects of

- Logic-Based Programming, Deductive Databases, Concurrent Constraints,
- Production Rules, Event-Condition-Action Rules, Business Rules,
- Multiset-, Term- and Graph-Rewriting, and Petri Nets

can be covered by suitable fragments of CHR.

Part III analyzes the programs from the CHR tutorial and a number of larger programs in more detail and more formally. The programs solve problems over finite and infinite domains of values: propositional satisfaction problems (Boolean Algebra), syntactic equations over rational trees and linear polynomial equations, implement the graph-based constraint algorithms of arc and path consistency, and the global lexicographic order constraint. We also directly implement description logic (extended with rules), which

is the formal basis of ontology languages of the semantic web. We give a program for the classical union-find algorithm with optimal time and space complexity. We parallelize the algorithm and generalize it for efficient equation solving. We use it in an efficient syntactic equation solver. All programs in this part are elegant, concise and effective.

The book ends with an extensive bibliography and an index.

Further Information and Software

The web-pages of this book offers teaching material such as slides and further exercises along with many links. It can be found via the comprehensive CHR website at `chr.informatik.uni-ulm.de`. The CHR site features access to research papers, software for download, programming examples, and descriptions of applications and projects. More than 1000 papers mentioning CHR are listed, many of them with links. There are lists of selected papers, ordered by topic, recency, and author. Tutorial notes and slides can be found as well.

More than a dozen free implementations of CHR exist. They are available in most Prolog systems, several in Haskell, and in more mainstream programming languages such as Java and C. Many can be downloaded for free from the CHR website. CHR is also available as WebCHR for online experimentation with dozens of example programs, including most from this book. So you can try out CHR from work, home or any internet cafe. Last but not least there is the mailing list `CHR@listserv.cc.kuleuven.ac.be` for beginners questions, discussion and announcements concerning CHR.

Acknowledgments or How CHR Came About

I came up with CHR during the first weeks at the European Computer Industry Research Centre in Munich in January 1991. Soon, Pascal Brisset implemented the first CHR compiler in Prolog. This was after an inspiring year with a Fulbright grant at SUNY at Stony Brook with David S. Warren, where I met Patreek Mishra and Michael Kifer, and after a research visit to Ehud Shapiro at the Weizmann Institute, where I met Moshe Vardi.

For the next five years or so, I got research papers introducing CHR rejected, even though there was some isolated interest and encouragement from people of the logic programming, constraint programming and term rewriting community.

Out of frustration I started to work on temporal reasoning until in 1995 Andreas Podelski invited me to contribute to the Spring School in Theoreti-

cal Computer Science in Chatillon with CHR. The breakthrough came when Slim Abdennadher provided the formal basis for the advanced semantics of CHR and essential CHR properties like confluence and operational equivalence and when Christian Holzbaur wrote an optimizing CHR compiler that would become the de-facto standard for a decade. All this cumulated in the invitation of Peter Stuckey to submit a CHR survey to a special issue of the Journal of Logic Programming in 1998 which became the main reference for CHR, leading to several hundred citations.

Since then, quite a few people contributed to the success of CHR, too many to thank them all by name, but let me just mention a few more of them. I was lucky again when Tom Schrijvers picked up CHR and within a few short years created the currently most active CHR research group at K.U. Leuven. He also edited special journal issues on CHR, organizes CHR workshops and maintains the CHR website.

I would also like to thank my Ph.D. students in Ulm, Marc Meister, Hariolf Betz, and Frank Raiser. They not only advanced the state of the art in CHR, they also helped me tremendously to deal with the downsides of academic life by sharing the burden. Together with Jon Sneyers and Ingi Sobhi, they provided detailed comments for parts of this book. I finally want to thank the reviewers of research papers that laid the ground for this book for their helpful comments.

Hariolf Betz pointed me to the Chinese character that became the CHR logo. CHR can not only be interpreted as an acronym for "Chinese HoRse". The Chinese character written "CHR" in the Yale transcription of Mandarin is derived from the character for horse but depending on the context, it can also mean *to speed, to propagate, to be famous.*

The Austrian artist Lena Knilli provided the art work for the book cover.

My first sabbatical semester gave me time to start this book. It would not have been written without public domain software such as the operating system Linux and Latex for type setting. It would not have been published in this form without the friendly people from Cambridge University Press. I doubt that I could have written the book at home or at my workplace. So special thanks to Oliver Freiwald, who gave me some work space in his insurance agency, to Marianne Steinert there and to the people from the Caritas social project coffee shop where during my sabbatical I had lunch and a lot of coffee, proofread the manuscript, and met my students for discussions.

Ulm, Germany, July 2006 - October 2008 **Thom Frühwirth**