



KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

EXTENSION AND OPTIMISING COMPILATION OF CONSTRAINT HANDLING RULES

Peter VAN WEERT

Dissertation presented in
partial fulfilment of the
requirements for the degree
of Doctor in Engineering

May 2010

EXTENSION AND OPTIMISING COMPILATION OF CONSTRAINT HANDLING RULES

Peter VAN WEERT

Jury:

Prof. Dr. ir. Hendrik Van Brussel, president

Prof. Dr. Bart Demoen, promotor

Prof. Dr. ir. Maurice Bruynooghe

Prof. Dr. ir. Thom Frühwirth (Universität Ulm)

Prof. Dr. ir. François Fages (INRIA Paris-Rocquencourt)

Dissertation presented in
partial fulfilment of the
requirements for the degree
of Doctor in Engineering

May 2010

© Katholieke Universiteit Leuven – Faculteit Ingenieurswetenschappen
Arenbergkasteel, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

Legal depot number D/2010/7515/58
ISBN number 978-94-6018-219-8

Abstract

Constraint Handling Rules (CHR) is a high-level declarative programming language based on multi-headed multiset rewrite rules, combined with aspects of logic and constraint programming. Originally designed for extending a host language with user-defined constraint solvers, CHR has evolved into a powerful, elegant general-purpose language with a wide spectrum of application domains.

The goal of this dissertation is to further improve the practical usability of the CHR programming language. As a first step, we therefore redesign the language's syntax, language features, and operational semantics to allow a more high-level, declarative programming style. Our streamlined CHR2 syntax allows for more natural, readable, and concise rule definitions. The operational semantics of CHR2 programs is designed to be as non-deterministic as possible, while still facilitating the effective execution control required for practical programming. In line with the 'what, not how' and 'algorithm = logic + control' maxims of declarative programming, the CHR2 system by default fully determines the execution strategy. When needed though, the programmer may control the order in which rules and conjunctions are executed using two orthogonal, familiar execution control constructs: rule priorities and sequential conjunction. Priorities are specified using symbolic priority constraints, which are more flexible than earlier proposals, and offer a better separation of logic and control.

We furthermore extended CHR with expressive language abstractions called aggregates. Aggregates are powerful, concise rule applicability conditions that collect information from larger parts of the constraint store. Well-known examples include `min`, `sum`, `count`, and `findall`. Our proposed framework supports nested aggregate expressions, efficient incremental aggregate computation and application-tailored user-defined aggregates. Aggregates eliminate the need for low-level encodings of aggregate computations commonly found in CHR programs. The extended CHR language thus fully regains its high-level, declarative nature.

A next crucial aspect of the practical usability of any programming language is the performance of its implementations. Because CHR2 rules are written at a very high level of abstraction, uncovering the optimal low-level execution steps required to evaluate them is very challenging. In the final part of the dissertation,

we therefore introduce, evaluate and refine many new and existing analyses and optimisation techniques for CHR programs. Two instances are discussed in more detail: We revise CHR's compilation scheme to optimise the space consumption of recursive programs, and develop novel techniques for optimal—both in space and in time—reapplication prevention of CHR propagation rules.

Lastly, for CHR to be really useful for practical applications, CHR must be embedded in a mainstream host language. We therefore developed K.U.Leuven JCHR, a state-of-the-art CHR system for Java. The thesis addresses both the language design issues of integrating CHR with imperative host languages, and the technical challenges faced when compiling CHR to imperative languages. JCHR is currently one of the most complete and efficient CHR implementations available, typically outperforming other rule-based systems by several orders of magnitude. The next-generation JCHR2 system moreover is a first reference implementation of the improved CHR2 language, extended with negation as absence.

Beknopte Samenvatting

Constraint Handling Rules (CHR) is een hoog-niveau declaratieve programmeertaal gebaseerd op meerhoofdige multiset-herschrijfgeregels, gecombineerd met aspecten uit logische en constraint-gebaseerde programmeertalen. Hoewel oorspronkelijk ontworpen voor het uitbreiden van bestaande gasttalen met gebruikersgedefinieerde constraint solvers, is CHR in het voorbije decennium geëvolueerd tot een krachtige, elegante, algemeen bruikbare programmeertaal, met een breed spectrum van toepassingsdomeinen.

Deze verhandeling heeft tot doel de praktische bruikbaarheid van CHR als programmeertaal verder te verbeteren. In een eerste stap, herontwerpen we de syntax, taalelementen, en operationele semantiek van de taal, teneinde een meer hoog-niveau, declaratieve programmeerstijl te ondersteunen. Onze gestroomlijnde CHR² syntax resulteert in meer natuurlijke, leesbare, en compacte regeldefinities. De operationele semantiek van CHR² programma's is ontworpen om zo non-deterministisch mogelijk te zijn, maar toch nog de effectieve uitvoeringscontrole te bieden die onontbeerlijk is voor praktisch programmeren. In overeenstemming met de 'wat, niet hoe' en 'algoritme = logica + controle' motto's van declaratief programmeren, bepaalt het CHR² systeem normaal gezien zelf volledig de uitvoeringsstrategie. Echter, wanneer nodig kan de programmeur de uitvoeringsvolgorde van de regels en conjuncties bijsturen met een combinatie van twee orthogonale, vertrouwde mechanismes voor uitvoeringscontrole: regelprioriteiten en sequentiële conjuncties. Prioriteiten worden gespecificeerd via symbolische beperkingen (constraints), die flexibeler zijn dan eerdere voorstellen, en een betere scheiding bieden van de logische en controle-aspecten van een programma.

Verder breiden we de CHR taal uit met expressieve taalabstracties genaamd aggregaten. Aggregaten zijn krachtige, bondige regeltoepasbaarheidsvoorwaarden die informatie verzamelen over grotere delen van de constraint store. Vertrouwde voorbeelden zijn onder andere `min`, `sum`, `count`, en `findall`. Het voorgestelde raamwerk ondersteunt geneste aggregaatsuitdrukkingen, efficiënte incrementele berekening van aggregaten, en toepassings specifieke, gebruikersgedefinieerde aggregaten. Dankzij aggregaten worden typische laag-niveau encodings overbodig, waardoor CHR zijn hoog-niveau, declaratief karakter herwint.

Een volgend cruciaal aspect van de praktische bruikbaarheid van een programmeertaal is de performantie van haar implementaties. Omdat CHR² regels geschreven zijn op een zeer hoog niveau van abstractie, is het bepalen van de optimale laag-niveau uitvoeringsstappen om deze regels te evalueren bijzonder uitdagend. Het laatste deel van de verhandeling introduceert, evalueert en verfijnt daarom nieuwe en bestaande analyse- en optimalisatietechnieken voor CHR programma's. Twee problemen worden meer gedetailleerd behandeld: We herwerken het compilatieschema voor CHR om het ruimtegebruik van recursieve programs te optimaliseren, en ontwerpen nieuwe technieken om op een optimale manier—zowel qua ruimte als tijd—te voorkomen dat zogenaamde propagatieregels meerdere malen worden toegepast.

Om CHR echt nuttig bruikbaar te maken voor praktische toepassingen, moet CHR ingebed worden in een mainstream gasttaal. Daarom hebben we K.U.Leuven JCHR ontwikkeld, een state-of-the-art CHR systeem voor Java. De thesis behandelt zowel taalontwikkelingsaspecten gerelateerd aan de integratie van CHR met imperatieve gasttalen, alsook de technische uitdagingen die opdagen bij het compileren van CHR naar dergelijke talen. JCHR is momenteel een van de meest complete en efficiënte CHR implementaties beschikbaar, en is typisch meerdere grootteordes efficiënter dan andere regelgebaseerde systemen. Ons JCHR² systeem is bovendien een eerste referentie-implementatie van de verbeterde CHR² taal, uitgebreid met negation as absence.

Contents

Abstract	iii
Beknopte Samenvatting	v
Contents	vii
List of Figures	xiii
List of Tables	xv
List of Listings	xvii
List of Symbols	xix
Acknowledgements	xxi
1 Introduction	1
1.1 Declarative Programming	1
1.2 Constraint Handling Rules	2
1.3 Goals and Overview	3
1.3.1 Part I — Background	3
1.3.2 Part II — CHR Language Design	4
1.3.3 Part III — Optimising Implementation of CHR	5
1.4 Bibliographic Notes	5
I Background	7
2 Rule-based Programming	9
2.1 Rules in Programming	9
2.2 Production Rules	10

2.2.1	Introduction	10
2.2.2	Historical overview	13
2.2.3	Matching algorithms	15
3	Logic and Constraint Programming	19
3.1	Logic Programming	19
3.1.1	Basics of logic programming	20
3.1.2	Operational semantics	22
3.1.3	Prolog: programming in logic	24
3.1.4	Conclusion	26
3.2	Constraint Programming	26
4	Constraint Handling Rules	29
4.1	Introduction	30
4.1.1	CHR(\mathcal{H})	30
4.1.2	Syntax	30
4.1.3	CHR by example	31
4.1.4	Relation to other formalisms	34
4.2	Formal Semantics	35
4.2.1	Logical semantics	36
4.2.2	The theoretical operational semantics ω_t	38
4.2.3	The refined operational semantics ω_r	40
4.3	Program Properties and Analysis	44
4.3.1	Termination	44
4.3.2	Confluence	44
4.3.3	Complexity	45
4.4	Language Extensions	47
4.4.1	Probabilities	47
4.4.2	Priorities	47
4.4.3	Adaptive CHR	48
4.4.4	Disjunction and search	49
4.5	Systems and Implementation	50
4.5.1	CHR(LP)	50
4.5.2	CHR(FP)	52
4.5.3	CHR(Java) and CHR(C)	53
4.5.4	Programming Environments	54
4.6	Applications	55
4.6.1	Constraint solvers	55
4.6.2	Algorithms	57
4.6.3	Programming language development	57
4.6.4	Industrial CHR use	60

II	CHR Language Design	61
5	A Next Generation CHR Language	63
5.1	Basic Building Blocks	64
5.1.1	Rule conditions	64
5.1.2	Constraint identifiers	65
5.1.3	Constraint arguments	66
5.1.4	Priority constraints	66
5.1.5	Batch and sequential conjunctions	69
5.1.6	Set semantics	71
5.1.7	Functional dependencies	72
5.2	Operational Semantics	74
5.2.1	Program normalisation	74
5.2.2	The operational semantics ω_2	75
5.2.3	Compatibility with other semantics	77
5.2.4	Discussion	81
5.3	Conclusions	82
6	Aggregates	83
6.1	Motivation	84
6.1.1	Negation as absence	84
6.1.2	Aggregates	87
6.2	Extensible Aggregate Framework	89
6.2.1	Universal aggregate construct	90
6.2.2	Common aggregates	92
6.3	Language Design	93
6.3.1	Aggregates for rule-based programs	94
6.3.2	Aggregates in CHR and CHR2	99
6.4	Formal Semantics and Properties	100
6.4.1	Operational semantics	100
6.4.2	Logical semantics and formal properties	101
6.5	Expressiveness case studies	102
6.6	Related Work	106
6.7	Conclusions	108
6.7.1	Future work	108
7	CHR for Imperative Host Languages	109
7.1	Impedance Mismatch	110
7.1.1	(C)LP language features	110
7.1.2	Imperative language features	112
7.2	Integrated CHR(<i>imperative</i>) Systems	113
7.2.1	Design philosophy	114
7.2.2	Arbitrary built-in constraints and solvers	115

7.3	The K.U.Leuven JCHR Systems	117
7.3.1	Historical overview	117
7.3.2	JCHR handlers	118
7.3.3	JCHR constraints	121
7.3.4	An integrated CHR(Java) system	122
7.3.5	Built-in constraints and solvers	123
7.3.6	Using a JCHR handler	125
7.3.7	Applications	127
7.4	Related work	128
7.4.1	CHR in C	128
7.4.2	CHR in Java	130
7.4.3	CHR in functional languages	130
7.4.4	Production Rule Systems	131

III Optimising Implementation of CHR 133

8 Optimising Compilation and Lazy Evaluation 137

8.1	Core language and normal form	138
8.2	Basic Compilation Methodology	140
8.2.1	Principal data structures and operations	140
8.2.1.1	The constraint store	140
8.2.1.2	Constraint iterators	140
8.2.1.3	The propagation history	141
8.2.2	Basic compilation scheme	142
8.2.3	Extension with negation	145
8.2.4	Extension with priorities	146
8.3	Program Analysis and Optimisation	147
8.3.1	Constraint invariants	148
8.3.1.1	Deriving constraint invariants	148
8.3.1.2	Unenforced constraint invariants	149
8.3.2	Optimising join computation	149
8.3.2.1	Loop-invariant code motion	150
8.3.2.2	Constraint indexing	151
8.3.2.3	Exploiting constraint invariants	152
8.3.2.4	Pre-commit backjumping	153
8.3.2.5	Post-commit backjumping	153
8.3.2.6	Fragile iterators	154
8.3.2.7	Join ordering	155
8.3.3	Reducing constraint store overhead	157
8.3.3.1	Late indexing	157
8.3.3.2	Late allocation	159

8.3.3.3	In-place and delayed modifications	159
8.3.3.4	Lazy indexing	159
8.3.4	Optimising constraint activation	160
8.3.4.1	Removal preference	160
8.3.4.2	Reducing schedule overhead	160
8.3.4.3	Passive removals	160
8.3.4.4	Passive occurrences	162
8.3.4.5	Dynamic passive occurrences	163
8.3.5	Optimising constraint reactivation	164
8.3.5.1	Selective reactivation	164
8.3.5.2	Delay avoidance	165
8.3.5.3	Generation optimisation	165
8.3.6	Program specialisation	165
8.3.6.1	Constraint specialisation	166
8.3.6.2	Guard simplification	166
8.4	Evaluation	166
8.4.1	CHR systems	166
8.4.2	Production rule systems	167
8.5	Discussion and Related Work	168
8.5.1	CHR systems	168
8.5.2	Production rule systems	169
8.6	Ongoing and Future Work	172
8.6.1	Dynamic optimisations	173
8.6.2	Global optimisations	174
9	Recursion Optimisations	175
9.1	Sequential Conjunctions and Recursion	176
9.1.1	Basic compilation scheme	176
9.1.2	Problem analysis: recursion and stack overflows	177
9.1.3	Problem demonstration: empirical results	179
9.2	Recursion Optimisations	180
9.2.1	Trampoline-based execution	180
9.2.2	Explicit stack	181
9.3	Evaluation	183
9.4	Conclusions	184
9.4.1	Related work	185
9.4.2	Future work	186
10	Optimising Propagation Rules	187
10.1	Propagation History Implementation	188
10.1.1	Optimising history maintenance	189
10.2	Non-reactive Propagation Rules	190

10.2.1	Introduction: from fixed to non-reactive CHR	190
10.2.2	Propagation history elimination	192
10.2.3	Optimised reapplication avoidance	195
10.3	Idempotence	198
10.3.1	Deriving idempotence	200
10.4	Evaluation	202
10.5	Conclusions	204
10.5.1	Related work	205
10.5.2	Future work	205
11	Conclusions	207
11.1	Contributions	207
11.1.1	CHR language design	207
11.1.2	Optimising implementation of CHR	208
11.2	Future Work	209
	Appendices	211
	A Heuristics for an A* Join Ordering Algorithm	213
	B Anti-Monotony-based Delay Avoidance	217
	C Benchmarks	221
	Bibliography	225
	Biography	245
	List of Publications	247

List of Figures

4.1	Transition rules of the theoretical operational semantics ω_t . . .	39
4.2	Transition rules of the refined operational semantics ω_r	42
4.3	The Apply transition rule of the priority semantics ω_p	48
4.4	A timeline of CHR implementations	51
5.1	Transition rules of the operational semantics ω_2	76
8.1	Performance comparison for two famous benchmarks	169
9.1	Execution of recursive CHR rules	177
10.1	Optimised reapplication avoidance for non-reactive rules	197

List of Tables

2.1	Rule-based programming terminology	18
4.1	Comparison of CHR with related formalisms	35
6.1	Predefined aggregates	92
6.2	Expressivity gained by using aggregates	105
7.1	Version history of the K.U.Leuven JCHR System	119
7.2	JCHR's Java built-in constraints	124
8.1	Performance comparison of state-of-the-art CHR systems	167
8.2	Performance comparison with production rule systems	168
8.3	Optimisations implemented by state-of-the-art CHR systems	170
9.1	Recursion limits for different CHR systems.	179
9.2	Recursion limits for standard CHR benchmark programs	180
9.3	Recursion optimisations	184
10.1	Benchmark results for two-headed rules	202
10.2	Benchmark results for non-reactive CHR rules	203
10.3	Benchmark results for idempotent propagation rules	203
C.1	Software versions used for benchmarking	221
C.2	Benchmark descriptions	224

List of Listings

2.1	OPS5 encoding of a rule from the WALTZ program	12
2.2	CLIPS/Jess encoding of a rule from the WALTZ program	12
2.3	Drools 5 encoding of a rule from the WALTZ program	12
3.1	Example Prolog program	21
4.1	The LEQ handler	32
4.2	Linearised form of the LEQ handler	32
4.3	The PRIMES handler	34
4.4	Occurrence numbering for the PRIMES handler	41
6.1	Dijkstra's algorithm with and without aggregates	103
6.2	Checking eulerianity in CHR with and without aggregates	103
6.3	Hopcroft's algorithm in CHR with and without aggregates	104
6.4	A Sudoku solver in CHR with and without aggregates	105
7.1	A JCHR implementation of the LEQ handler	120
7.2	A JCHR2 encoding of the MERGESORT handler	121
7.3	A JCHR2 implementation of the GCD handler	122
7.4	Declaration of built-in equality solvers in JCHR	125
7.5	Using a JCHR handler from Java	126
7.6	A CCHR implementation of the LEQ handler	129
8.1	Basic compilation scheme for a positive occurrence	143
8.2	Naive compilation using the basic compilation scheme	144
8.3	Basic compilation scheme for a negated conjunction	145
8.4	Constraint activation, extended to deal with priorities	147
8.5	Loop-invariant code motion	150
8.6	Indexing and exploiting constraint invariants	152

8.7	Optimal join computation	156
8.8	Passive modification in a rule from WALTZ	161
10.1	The FIBBO program	191
10.2	The FIB program	191
10.3	Bank account example	193

List of Symbols

Symbol	Description	Page
θ	(matching) substitution / variable renaming	20
$\theta(X)$	instantiation of X after applying substitution θ	20
$vars(X)$	the variables occurring in X	39
$\bar{\forall}X$	$\forall x_1, \dots, \forall x_n X$ with $\{x_1, \dots, x_n\} = vars(X)$	36
$\bar{\exists}_Y X$	$\exists x_1, \dots, \exists x_n X$ with $\{x_1, \dots, x_n\} = vars(X) \setminus vars(Y)$	39
$\pi_V(X)$	$\exists x_1, \dots, \exists x_n X$ with $\{x_1, \dots, x_n\} = vars(Y) \setminus V$	43
$++$	sequence concatenation	39
\uplus	multiset union	39
\sqcup	disjoint union: $\bar{\forall}(X = Y \sqcup Z \leftrightarrow X = Y \cup Z \wedge Y \cap Z = \emptyset)$	39
$[e_1, \dots, e_n]$	sequence of elements e_1, \dots, e_n	40
$X[i]$	the i th element in sequence X (starting from 1)	40
$ X $	the size of a set X (or length of a sequence X)	40
$[H T]$	$[H] ++ T$	42
c/n	a predicate or constraint c and n arguments	21
\mathcal{P}	a CHR program	33
\mathcal{P}^*	a normalised CHR program	33
\mathcal{H}	host language	30
$\mathcal{D}_{\mathcal{H}}$	built-in constraint theory	36
$\text{CHR}(\mathcal{H})$	CHR with host language \mathcal{H}	30
CHR^{P}	CHR with rule priorities	47
CHR^{\vee}	CHR with disjunctive rule bodies	49
$\text{CHR}\wp$	the next generation CHR language defined in this thesis	63

Symbol	Description	Page
ω_t	the theoretical operational semantics of CHR	38
ω_r	the refined operational semantics of CHR	40
ω_p	the priority semantics of CHR ^{FP}	48
ω_{rp}	the refined priority semantics of CHR ^{FP}	48
ω_∂	the operational semantics of CHR ₂	75
$c\#i$	an identified CHR constraint c with identifier i	38
$c\#i:j$	an occurred identified constraint $c\#i$ being matched with its j th occurrence	41
$\text{CHR}(c\#i)$	returns the CHR constraint c	38
$\text{ID}(c\#i)$	returns the CHR constraint identifier i	38
\mathbb{G}	goal (of a CHR state)	38
\mathbb{A}	activation stack	41
\mathbb{S}	CHR constraint store	38
\mathbb{B}	built-in constraint store	38
\mathbb{T}	propagation history	38
$\mapsto_{\mathcal{P}}$	transition between two states in program \mathcal{P}	39
$\mapsto_{\mathcal{P}}^*$	finite sequence of transitions in program \mathcal{P}	40

Acknowledgements

I can no other answer make, but, thanks, and thanks.

— **William Shakespeare** (1564–1616)
English poet and playwright

This dissertation presents the main results of my Ph.D. research carried out at the Computer Science department of the K.U.Leuven during the past four and a half year. Its completion would have been impossible without the help and support of many people to whom I would first like to express my sincere gratitude.

Let me start with the time leading up to the start of my thesis project. First of all, I owe a great deal of gratitude to Bart Demoen and especially Tom Schrijvers for their excellent guidance during my master’s thesis on “Constraint Programming in Java: a user-friendly, flexible and efficient CHR-system for Java”. Not only did they introduce me to the central topics of this dissertation—Constraint Handling Rules, programming language design and optimising compilation—they also showed me that research can be fun, challenging, and rewarding. After I submitted the thesis, they moreover invited me to write my first paper, and to present my work at the CHR 2005 workshop in Sitges. This recognition was an important motivation for me to start as a Ph.D. student the next year.

For a brief time, my career as a Ph.D. student actually started in the SecLang taskforce of the DistriNet research group, under the supervision of Frank Piessens. Thank you, Frank, for the confidence and for acting as my supervisor, even if it was only for four months, and especially for being so understanding when we decided it was better to switch. It was by no means a fault of yours, I simply was unable to let go of my master’s thesis: the optimising compilation of JCHR had become almost a hobby, and the perfectionist in me simply could not shake the feeling there was still too much room for improvement there.

Thanks also to my officemates of old, Kristof Geebelen, Thomas Heyman, Eryk Kulikowski, and Koen Yskout, for making those first few months at the department so enjoyable. Our ‘basement’ office, at the time conveniently located next to the ping pong table, was no doubt the most lively office I have worked in at the department. Furthermore, I must thank them and all other members of the

DistriNet Alma gang, Koen Buyens, Kris Demarsin, Bart Elen, Johan Gregoire, Riccardo Scandariato, Tom Stijnen, Yves Younan, Koen Victor, Kim Wuyts, etc., for making our daily Alma trips a treat, even long after I left DistriNet. I almost regret kicking the Alma diet when I think back to those days.

The research reported in this dissertation finally took off in January 2006 when I joined the DTAI research group to further pursue the research I started there as a master's student. A special thanks to my supervisor Bart Demoen for taking me back. I can honestly say that I cannot imagine a wiser, cleverer, more inspirational, devoted, or more approachable mentor. Thanks for always being there when I needed help or guidance, and for encouraging me to find my own way in my Ph.D. research. Admittedly, it took me quite a while to find it, but in the end it all turned out well.

As with my thesis topic, it took me some time to settle down into my final office as well. During the first months as a DTAI member, I moved around the first floor quite a bit. Even though I did not get to know half of you half as well as I should like, thanks Manh Thang Nguyen (rest in peace, my dear friend), Quan Phan, Álvaro Cortés, Johan Wittcox, Hou Ping, Remco Tronçon, Qiang Fu, and anyone else I may have shared an office with, however briefly, for your pleasant company.

Work never felt quite like home though until Leslie De Koninck and Jon Sneyers moved from across the hall into room 01.05 (later labelled 01.171, and now 01.167) to replace Remco and Fu. I cannot thank them enough for the relaxed, inspiring and productive atmosphere that always permeated our CHR bastion. I enjoyed tremendously our successful collaborations, the intense discussions at the white board, and of course our daily coffee breaks. Even though I do not drink coffee, these trips to the cafeteria were indispensable moments of relaxation and socialising. I am also grateful to Jon for allowing me to jettison my guilt for being somewhat lazy, as he is always so kind as to keeping his desk in an even more chaotic state than mine.

In April 2009, after completing his excellent Ph.D. in a record-breaking three and a half year, Leslie left to carry out his life-long dream of living in Australia. Soon after, being abandoned by his office mate Tom Schrijvers¹, Pieter Wuille stepped up to the difficult task of replacing Leslie. After a trial period of six months, we decided he could stay. With Pieter there, I also no longer was the only one haunted by the universal laws of implementation (namely Hofstadter's Law² and the ninety-ninety rule³). You can also always count on Pieter for citing intriguing engineering trivia and movie quotes, and for computing the most interesting facts. Thanks also for teaching us that we can also take the stairs for

¹Tom visited the Cambridge University Computer Laboratory for six months.

²Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

³The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time.

our coffee breaks. We have since gone up and down about 53,125 stairs, more than 10km height difference, allowing me to burn 8,500 calories or more than 29 sausage rolls. This is still only about 36% of the rolls I have eaten—thank you Alma for your sausage rolls—but a good start nevertheless.

One of the most interesting rewards of doing research and writing papers is that it gives you the opportunity to see the world. Forever engraved in my memory are the confusing maze of alleyways and canals in football crazy Venice, the port cellars in Porto, the rain in Udine, and the cocktail and karaoke bars in Pasadena. Thanks to all those who made these trips unforgettable: the colleagues from Leuven that joined me—my fellow CHR team members, Theofrastos Mantadelis, Quan Phan, Hanne Vlaeminck, etc.—as well as the many new friends I met on these travels. A honorary mention goes to the countless wonderful people I met at the Constraint Programming summer schools in Lloret de Mar and St. Andrews. A special thanks also to Paolo Pillozzi, for driving me across Slovenia (together with Leslie) and later America (together with Dean Voets in our ever-so-cool Ford Mustang), allowing me to gaze once more upon the great canyons of the West—Zion, Bryce, and the ‘big hole’ itself. And who could forget the good times, and not-so good pizza’s, we had in Vegas!

A big thanks to all the members of the CHR community—Hariolf Betz, Gregory Duck, Mark Meister, Frank Raiser, Jairson Vitorino, and many more—for making the CHR workshop, as well as all the social events surrounding them, an event to look forward to each year. To everyone who has used my JCHR system, and to Slim Abdennadher and Shehab Alaa El-Din Fawzy for building the Eclipse plug-in: your input and feedback has been invaluable. Martin Sulzmann, I cannot thank you enough for your catching enthusiasm, and for taking an interest in my research and inviting me to stay in Singapore for a month. Thanks Martin, Edmund Soon Lee Lam, and Kenny Lu Zhuo Ming, for your warm welcome, for the many hours discussing concurrency issues, and for allowing me to sample the diverse tastes and cultures Asia has to offer. A special thanks, finally, to Thom Frühwirth, for inventing CHR, for forever propagating his passion for the language, for closely following my work ever since the CHR 2005 workshop, for the stay in the lovely city of Ulm during the CHR Working Week, and for inviting me to co-organise this year’s CHR workshop and summer school.

Next to research, a Ph.D. student at our department is responsible for teaching exercise classes. I have always enjoyed doing this tremendously. Thanks to all my students and colleagues; especially Henk Olivie, whose unsurpassed passion for the teaching profession certainly rubbed off to all the members of the dedicated BVP team—Erik Boij, Nik Corthaut, Pieter-Jan Drouillon, Joris Klerkx, Paolo Pillozzi, Stefan Raeymaekers, Hanne Vlaeminck, etc.—during the weekly meetings.

The completion of this thesis would furthermore have been impossible without the help and support of many others outside the department. Firstly, financially, I am forever indebted to the Research Foundation Flanders (FWO Vlaanderen),

for funding me for four years as a FWO research assistant. Secondly, and more importantly, my sincere gratitude goes to all my friends, family, and parents. My friends—Anne-Katrien, Bert, Geert, Geert & Marijke, Kris & Isabel, Pieter & Eveline, Sven & Femke, just to name a few of my fabulous friends here in Leuven—thanks for making Leuven my second home, for the West Coast Road Trip, the weekends in the Ardennes, the relaxing times in France and Switzerland, and so much more. My family, for the yearly hiking trips to the Alps, the weekends in the Ardennes, for doing at the very least an excellent job at pretending you enjoy my quizzes, and so much more. My parents and brother, for introducing me to the world of computers, for the unconditional love and support, for putting up with me when I was stressing about deadlines, for driving me back and forth to Leuven, and so much more. Thanks all for making all the time outside of the office at least as enjoyable!

To conclude, I would like to thank to the members of the jury, for being the first to read my text, and for your insightful comments. And finally, thank you reader, for reading these acknowledgements at least. I hope you enjoy reading my dissertation as much as I enjoyed preparing and writing it.

Chapter 1

Introduction

Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

— Eugene C. Freuder (1997, *Constraints* 2(1):57–61)

1.1 Declarative Programming

Conventional programming languages mostly fall under the umbrella term *imperative languages*. In natural languages, the imperative mood expresses direct commands or requests. Imperative computer programs similarly specify sequences of commands for the computer to perform. Much like a cooking recipe, an imperative program specifies all consecutive steps required to produce some desired result from given input data. Even though the steps have become more high level, and powerful mechanisms have been developed to structure, group, and reuse different sequences of steps, most modern-day mainstream programming languages remain imperative by nature.

The ‘Holy Grail’ of *declarative programming*¹, on the other hand, is that a program should only describe *what* the problem is, rather than *how* to solve it. Using some intuitive, high-level formalism, a programmer indicates only the essential characteristics of the problem (or the required solution); the computer

¹The notion ‘declarative programming’ has several definitions. It is therefore debatable whether or not languages such as Constraint Handling Rules or Prolog are ‘declarative’. The more puristic definition requires declarative programs to have a strict correspondence with a theory in a suited mathematical logic. As introduced further in this section, in this dissertation we employ the common, more pragmatic view where declarative languages are high-level languages, contrasted with imperative ones based on a defining ‘what, not how’ principle.

then determines the best way to solve it. To use another cooking metaphor: you only describe to your Personal Cook (PC) the desired properties of your favourite dish, you do not want to have to spell out the recipe.

The advantages of declarative programming are multifold. Software development time is reduced, even for complex problems for which the solution algorithm is far from apparent. Declarative programs are generally more concise and readable, making them easier to adapt, maintain, and reason about. Their semantics (meaning) is well-defined and intuitive, often founded by some mathematical or logical formalism (see further). Even without knowing how the underlying implementation actually solves your problem, this facilitates debugging and (automatic) verification of correctness properties. Moreover, domain experts typically spend a lot of research and effort into the design and implementation of the algorithms and compilers used to execute declarative programs. The result is that declarative programs often are at least as efficient and scalable as programs written in conventional languages.

Declarative languages differ on the formalism used to specify programs. The three most well-known categories are:

Functional programming languages are based on the evaluation of mathematical functions. Examples are LISP, ML, Erlang, and Haskell.

Logic programming languages have their roots in mathematical logic. The best-known logic programming language is Prolog, but many others exist.

Constraint programming languages employ various types of mathematical relations between program variables to constrain candidate solutions. Constraints are typically embedded in logic or imperative languages via constraint solving libraries (such as ECLⁱPS^e or Gecode).

Logic and constraint programming are introduced in more detail in Chapter 3.

Of particular interest to us is yet another class of declarative programming languages, commonly referred to as *rule-based programming languages*. Rule-based programs consist of high-level, logical if-then rules, and are the descendants of so-called *production rule* languages, which were popular in the specification of AI applications in the 1970s and 1980s. Today, interest in rule-based programming languages is revived due to the growing proliferation of software solutions based on *business rules*. A proper introduction of the rule-based languages paradigm is found in Chapter 2.

1.2 Constraint Handling Rules

Constraint Handling Rules (CHR) is a rule-based programming language, commonly embedded in an existing host language. It elegantly combines aspects from rule-based, logic and constraint programming.

CHR was originally created by Thom Frühwirth in 1991 as a special-purpose language for the high-level specification of constraint solvers. In constraint programming, constraint solvers are the software engines that solve the problems specified in terms of constraints. CHR extends the host language with new, user-defined constraints by means of a series of logical rules that repeatedly rewrite these constraints to simpler forms until a solution is found.

The theory and practice of CHR became the subject of an active niche research field. By now, the theoretical foundations, properties and analyses of CHR are well understood, and dozens of efficient implementations of CHR exist for Prolog, Java, Haskell, etc. It has also become clear that CHR is useful as a general-purpose declarative programming language. This is witnessed by its countless applications in a wide range of areas, including multi-agent systems, type system design, and natural language processing. This evolution has only heightened the need for an even better understanding of CHR programs, and for even more practical, robust, and efficient CHR systems.

A more thorough introduction to CHR is given in Chapter 4.

1.3 Goals and Overview

The main goal of this dissertation is to improve the practical usability of the CHR programming language. This goal is actually threefold:

1. ensure that CHR programs can be written effectively and intuitively using a high-level, declarative programming style
2. ensure that these programs are executed very efficiently
3. integrate CHR with mainstream (imperative) programming languages, facilitating the use of CHR in real-life applications

We now provide an overview of the more specific goals of the different parts of this dissertation. After a relatively broad background part, our main contributions are presented in six chapters, equally distributed over two parts: *CHR Language Design* and *Optimising Implementation of CHR*. Afterwards, a final chapter offers general conclusions, as well as a perspective on important future research challenges.

1.3.1 Part I — Background

We find it imperative that this dissertation is accessible to readers without a background in either rule-based programming or constraint logic programming. In Chapter 2, we therefore first introduce rule-based programming, and the main algorithms used for their implementation. Chapter 3 similarly establishes several basic concepts and terminology from logic and constraint programming.

Chapter 4 is a more comprehensive overview of the CHR programming language. We cover many aspects, ranging from theory to practice. This includes CHR’s basic syntax and distinguishing characteristics, the different logical and operational semantics of the language, important program properties, existing language extensions and implementations, and the wide range of recent CHR-based applications. In a way, this chapter already reflects a first contribution we made in the form of (Sneyers, Van Weert, Schrijvers, and De Koninck 2010), a comprehensive survey of a decade of CHR research.

1.3.2 Part II — CHR Language Design

A critical assessment of the syntax, language features, and operational semantics of current systems and proposals quickly reveals that CHR’s expressiveness, usability and declarativeness are frequently lacking. CHR aims at supporting a very high-level, declarative programming style. In practice, however, systems typically abide an imperative-like operational semantics, and lack declarative means of exerting execution control. From times, CHR’s syntax is archaic and overly verbose, and common tasks require cumbersome low-level encodings that further nullify the advantages associated with declarative programming.

Based on this analysis, our goal is thus to design a modernised, streamlined CHR language that eliminates the disadvantages of current approaches, while integrating and further improving on their advantages. Chapter 5 outlines the core result of this effort, a next generation CHR language denoted CHR2. An important point of attention is the trade-off between execution control and declarativeness. Of course, an intuitive, predictable runtime behaviour combined with flexible execution control mechanisms is indispensable for practical programming. But the declarative ideal entails that control should only be exerted then and there where needed, as expressed by Kowalski’s maxim ‘Algorithm = Logic + Control’ (Kowalski 1979). By default, it should be the task of the compiler or runtime to decide the optimal execution strategy. Current approaches by Duck et al. (2004) and De Koninck (2008) fail to strike the right balance, but a judicious refinement and combination of their underlying ideas leads to a particularly elegant, practical language. We formally fix the operational semantics of the CHR2 language, and compare it in detail with aforementioned existing approaches.

In Chapter 6, we observe that for accumulating information from larger portions of the constraint store, CHR programmers are forced to resort to tedious low-level encodings, often cross-cutting the entire program. Practice learns that this impedes all declarative advantages of programming in CHR. We therefore investigate the extension of CHR2 with powerful language abstractions for negation and other aggregates such as `sum`, `min`, and `findall`.

Conventionally, CHR is embedded in Prolog. To maximise the language’s applicability, proper embeddings of CHR in mainstream imperative (object-

oriented) languages are required. We therefore developed the K.U.Leuven JCHR system for Java, which is currently one of the most complete and efficient CHR systems available. Its successor, JCHR2, moreover constitutes a first reference implementation of a substantial subset of CHR2. In Chapter 7, we outline and motivate the general system and language design issues and choices, focussing on a proper, natural integration of CHR and the imperative host.

1.3.3 Part III — Optimising Implementation of CHR

In the final part of the dissertation, we introduce, evaluate and refine new and existing program analyses and optimisation techniques for CHR programs. CHR rules are written at a very high level of abstraction. Uncovering the optimal low-level execution steps required to evaluate them is therefore very challenging, particularly when evaluating under the semantics of CHR2, which we purposely designed to be as nondeterministic as possible.

There is a vast literature on the efficient compilation of CHR, with many publications introducing different analyses and optimisations. Almost without exception, however, these are based on the nearly deterministic operational semantics of current systems, and targeted towards constraint logic host languages. Besides the foundational work of De Koninck (2008), the optimising implementation of recent extensions has received little or no attention either.

In Chapter 8, we provide a long overdue comprehensive compendium of CHR optimisations, ported to an imperative setting and fully extended to an expressive subset of CHR2. We put existing CHR compilation techniques in one accessible, coherent framework, and show how in the JCHR systems we have further refined, extended, and improved them. We also introduce several (often previously unpublished) optimisation techniques aimed at optimising both space and time performance.

Two important optimisations are discussed in more detail in separate chapters. Chapter 9 explains the issues we experienced with recursive programs, and why these are considerably aggravated when compiling to an imperative target language. Our redesigned compilation schemes are aimed at resolving these issues. Next, in Chapter 9, we fill an important void in CHR implementation research, namely the optimisation of the expensive propagation histories maintained for CHR propagation rules.

1.4 Bibliographic Notes

Most of the material in this dissertation is based on earlier publications, often joint work with other authors. Most chapters though have been considerably rewritten, corrected, or extended. In particular:

- Chapter 4 mostly consists of shortened versions of selected sections from our survey article (Sneyers, Van Weert, Schrijvers, and De Koninck 2010).
- Chapter 5 combines preliminary ideas from (Van Weert, De Koninck, and Sneyers 2009) and (Van Weert 2010a). They are considerably better motivated and worked out. The formal semantics in Section 5.2 replaces the earlier, incorrect version from (Van Weert et al. 2009); all formal compatibility results in this section are new as well.
- Chapter 6 integrates (Van Weert, Sneyers, Schrijvers, and Demoen 2006a) and (Sneyers, Van Weert, and Schrijvers 2007). Section 6.3 discusses many language design issues that were left out of these earlier presentations of this work. The chapter focusses on language design issues; our efficient source-to-source implementation of aggregates, presented in (Van Weert, Sneyers, and Demoen 2008), is only very briefly discussed.
- Chapter 7 is based on (Van Weert, Wuille, Schrijvers, and Demoen 2008). Among other things, the description of the JCHR and JCHR2 systems has been reworked considerably.
- Chapter 8 succeeds (Van Weert 2008a; Van Weert, Wuille, Schrijvers, and Demoen 2008; Van Weert 2010a), three earlier overviews of CHR compilation methodology (all written for a different intended audience). The new version is written to be even more comprehensive and clear, and also adds a lot of previously unpublished material: novel optimisations, extensions and improvements of existing techniques, etc. An extended presentation of our contributions on efficiently implementing join ordering algorithms based on Section 8.3.2.7 and Appendix A will appear as (Van Weert 2010b).
- Chapter 9 presents work published in (Van Weert 2008a) and (Van Weert, Wuille, Schrijvers, and Demoen 2008), but taking a more high-level view. It also demonstrates how to extend our optimised compilation techniques to rule-based languages with priorities.
- Chapter 10 combines (Van Weert 2008c) and (Van Weert 2008b). It adds some minor optimisations from JCHR, and a discussion on how to extend our history-related optimisations to richer rule-based languages.

In addition to the work presented in this text, we have also worked on the language design of actor-based programming languages (e.g. Erlang). We proposed an extension of such languages with CHR-inspired multi-headed, guarded receive patterns. We demonstrated their increased expressiveness, explored possible semantics, and implemented a reasonable first prototype. This work was presented in (Sulzmann, Lam, and Van Weert 2008).

Part I

Background

Chapter 2

Rule-based Programming

A few strong instincts and a few plain rules suffice us.

— **Ralph Waldo Emerson** (1803–1882)
US philosopher, poet, essayist

The term *rule-based programming* is commonly understood to denote the use of a family of programming languages that descend from so-called *production rule* languages. In essence, CHR is very similar to these rule-based programming languages. In Section 2.2, we therefore discuss their basic features and application areas, as well as the standard techniques used in their evaluation.

Rules, however, play a central role in many other programming languages and formalisms as well. We list some relevant examples in Section 2.1.

2.1 Rules in Programming

- Reduction systems are an important branch of computer science with applications e.g. in algebra, recursion theory, software engineering, and programming languages. Popular instances include string rewriting (Book and Otto 1993), term rewriting (Bezem, Klop, and de Vrijer 2003; Baader and Nipkow 2003), and graph rewriting (Ehrig and Rozenberg 1999). In these paradigms, rewrite rules specify how expressions of some formal language are transformed. Operationally, these rules exhaustively replace (sub)expressions with simpler, more canonical ones, until some normal form is reached. We briefly discuss their relation to CHR in Section 4.1.4.
- Active databases use ECA rules (Event Condition Action rules) to automatically respond to specific database events. Widom and Ceri (1996):

“Active database systems enhance traditional database functionality with powerful rule-processing capabilities, providing a uniform and efficient mechanism for many database system applications. Among these applications are integrity constraints, views, authorisation, statistics gathering, monitoring and alerting, knowledge-based systems, expert systems, and workflow management. ”

Starting from the early 1990s, there is a vast body of research on active databases. While ECA rules are mostly heavily inspired by production rules, both in syntax and evaluation strategies, they are governed by completely different design and implementation considerations. We therefore refer the interested reader to e.g. (McCarthy and Dayal 1989; Hanson and Widom 1993; Widom and Ceri 1996; Paton and Díaz 1999).

- Rules play an important role in theoretical computer science as well. Examples include production rules in formal grammars, and inference rules in logic and formal systems. Immediate practical applications include the various rule-based (grammar) formalisms used e.g. in the construction of parsers and compilers, and to model natural language in computational linguistics.
- In logic programming, as discussed in Chapter 3, programs consists of logical formulae or rules (typically Horn clauses).

With the apparent exception of ECA rules, CHR has been applied successfully to similar tasks as the formalisms listed above. Also, in CHR-related research, a major recent trend is a deeper, theoretical investigation of CHR’s relationship to such formalisms (see Section 4.1.4). This has led Frühwirth (2009) to propose CHR as a *lingua franca* for rule based programming.

2.2 Production Rules

Section 2.2.1 introduces *production rule systems*. These rule-based systems became popular as a tool to implement artificial intelligence applications, and, more recently, have evolved into one of the key technologies in the implementation of business rules. We provide a brief historical overview in Section 2.2.2. In Section 2.2.3, finally, we discuss the standard matching algorithms used by production rule engines.

2.2.1 Introduction

Modern production systems have a multitude of features. In this quick introduction, we restrict ourselves to the basic core features shared by all systems. We discuss some of the most important extensions in Section 2.2.2.

We adopt the terminology employed e.g. by CLIPS and Jess. Table 2.1 at the end of this chapter provides an overview of alternate terms used by other rule-based systems and formalisms, including CHR.

Working memory

The *working memory* (*WM*) of a traditional production system is a set of *facts*. Each fact is a runtime instance of a *template*. A template has a unique name, and declares a number of *slots*. Each attribute has a unique name and often also a type. Templates and facts are similar to classes and (immutable) objects in object-oriented languages, or tables and rows in a relational database. By default, once asserted, attribute values can no longer be changed (see the discussion of the `modify` action later).

Rules

A production rule program contains a number of condition-action rules. The left-hand side (LHS) of a rule is a conjunction of *condition elements* (*CEs*), specifying the conditions under which the rule is applicable. A *positive CE* specifies that a fact of a specified class must be present in the WM; a *negative CE* specifies it may not be present. CEs mostly also add a number of restrictions on the fact's attribute values. The right-hand side (RHS) is a conjunction of *actions*. The basic actions are `assert` and `retract` that, respectively, add and remove facts from the WM. Typically, a `modify` action is also supported, which acts as syntactic sugar for a `retract` followed by an `assert`.

Example 2.1. Listings 2.1–2.3 show an encoding of the same production in three different systems. The rule is taken from the famous WALTZ production rule program¹ which implements Waltz' seminal algorithm for interpreting line drawings of three-dimensional scenes (Waltz 1975). A directed edge between two points is represented as a fact of class `edge`. Besides the two points (represented using a single integer number), fact of class `edge` also have a boolean attribute `joined`. The rule detects a particular type of junctions between these edges. The first CE specifies a `stage` fact has to be present, whose `value` attribute indicates that the aim of the current execution stage is indeed detecting junctions. The remaining CEs specify that two edges starting from the same `base` point, but ending in different points, have to be present, and similarly that no third edge starting from that point may be present. If the rule is applied, or *fired*, the RHS specifies that a corresponding `junction` fact must be added, and the `joined` attribute of both matched `edge` facts set to `true`.

¹The WALTZ program is part of the so-called “Texas benchmark suite” (Miranker et al. 1991), so named after the University of Texas at Austin, the affiliation of its authors (Brant, Grose, Lofaso, and Miranker 1991). These benchmarks, particularly MANNERS and WALTZ, are still the de-facto standard benchmarks for production systems (IllationTM 2007).

Listing 2.1 The example WALTZ rule in OPS5 (source: Miranker et al. 1991)

```
(p make_L
  (stage ^value detect_junctions)
  (edge ^p1 <base> ^p2 <p2> ^joined false)
  (edge ^p1 <base> ^p2 {<p3> <> <p2>} ^joined false)
  -(edge ^p1 <base> ^p2 {<> <p2> <> <p3>})
-->
  (make junction ^type L ^base_point <base> ^p1 <p2> ^p2 <p3>)
  (modify 2 ^joined true)
  (modify 3 ^joined true)
)
```

Listing 2.2 The example WALTZ rule in CLIPS/Jess (source: Miranker et al. 1991)

```
(defrule make_L
  (stage (value detect_junctions))
  ?f2 <- (edge (p1 ?base) (p2 ?p2) (joined false))
  ?f3 <- (edge (p1 ?base) (p2 ?p3&~?p2) (joined false))
  (not (edge (p1 ?base) (p2 ~?p2&~?p3)))
=>
  (assert (junction (type L) (base_point ?base) (p1 ?p2) (p2 ?p3)))
  (modify ?f2 (joined true))
  (modify ?f3 (joined true))
)
```

Listing 2.3 The example WALTZ rule in Drools 5 (source: JBoss 2010). Note that in this encoding plain Java objects are used as facts.

```
rule "make L"
when
  Stage ( value == Stage.DETECT_JUNCTIONS )
  $edge1: Edge( $base:p1, $p2:p2, joined == false )
  $edge2: Edge( p1==$base, $p3:p2 != $p2, joined == false )
  not Edge( p1==$base, p2 != $p2, p2 != $p3 )
then
  insert( new Junction($p2, $p3, 0, $base, Junction.L) );
  modify( $edge1 ) { setJoined(true) }
  modify( $edge2 ) { setJoined(true) }
end
```

2.2.2 Historical overview

OPS5

Developed in the late 1970s by Forgy (1979), the OPS (“Official Production System”) family of languages were used primarily for applications in the areas of artificial intelligence, cognitive psychology, and expert systems. OPS5 was one of the first really successful production rule languages (Forgy 1981; Brownston, Farrell, Kant, and Martin 1985; Cooper and Wogrin 1988).

In the 1980s, production systems became the dominant knowledge representation methodology in *expert systems*. An expert system is a classic application of artificial intelligence: it emulates a human advisor, attempting to (interactively) provide answers to problems where normally human experts would need to be consulted. The high-level, non-procedural production rules facilitated the process of gathering and maintaining the necessary large *knowledge bases* from or even by domain experts. This process is often called *knowledge engineering*.

Example 2.2. The R1 (later called XCON, for eXpert CONfigurer) program was an expert system written in OPS4/OPS5 by McDermott (1980) to assist in the ordering of DEC’s VAX computer systems. R1 automatically selected the computer system components based on the customer’s requirements. The development of R1 followed two previous unsuccessful efforts to write an expert system for this task, in FORTRAN and BASIC.

R1 was the first commercially successful expert system. Four years after its initial deployment (Bachant and McDermott 1984), the program had over 3,300 rules, had processed over 80,000 orders, and achieved over 90% accuracy. It was estimated to be saving DEC millions of dollars a year by reducing the need to give customers free components when technicians made errors, by speeding the assembly process, and by increasing customer satisfaction (Kraft 1984). XCON eventually even surpassed 10,000 rules (McDermott 1994).

One of the key innovations of the OPS systems that allowed it to scale to large applications was the Rete algorithm used in their implementation (Forgy 1979, 1982). We discuss this in more detail in Section 2.2.3.

CLIPS

CLIPS (“C Language Integrated Production System”) was originally developed by NASA, from the mid 1980s until the mid 1990s. It was initially created because other available tools used LISP as the base language, which was deemed impractical for real applications. Over the years, CLIPS evolved from a prototype internally used by NASA, to a powerful expert system shell widely used throughout government, industry, and academia (Giarratano and Riley 1989). For a complete history of CLIPS, see (Riley et al. 2010). Today, CLIPS is maintained as public

domain software, independently from NASA. It remains one of the most efficient production rule engines available (IllationTM 2007).

CLIPS' syntax, expressive language constructs, and operational semantics, have heavily influenced subsequent production systems. Notable features include *saliency* (rule priorities), new CE types such as `forall` and `exists`, support for nested CEs, and object oriented extensions (Riley et al. 2008).

Jess

Jess started in 1995 as a reimplementaion of CLIPS in Java, but has since evolved to a powerful, popular expert shell for the Java platform (Friedman-Hill 2003; Friedman-Hill 2010). Particular contributions are the ability to directly manipulate and reason about regular Java objects, and the addition of so-called backwards chaining capabilities. Jess was used as the reference implementation for the JavaTM Rule Engine API (Selman et al. 2004).

Business Rules Management Systems

In recent years adoption of rule-based technologies has surged. Blaze Advisor (by Fair Isaac), ILOG JRules (now acquired by IBM), Haley Rules (now acquired by Oracle), and PegaRULES (by Pegasystems), are just a few examples of the numerous commercially successful rule-based systems available today. Production rules, now known as *business rules*², are an established technology in finance, banking and insurance, and companies all across the board are starting to use rules to maximise business agility. Modern-day *Business Rules Management Systems (BRMS)* are large, feature-rich software platforms. Complementing the traditional rule execution engine, they typically offer many additional software components. Their key merit is that multiple users of different skill levels, including non-technical business users, can access, monitor, edit, add, remove or deploy rules through various easy-to-use graphical user interfaces. This includes GUIs—and increasingly also web-based interfaces—where rules can be specified in near natural language, or by means of spreadsheets.

JBoss Drools

JBoss Drools (Bali 2009; Browne 2009) is probably the only open source, publically available BRMS that can rival with top commercial products in feature completeness. It is currently very actively developed. Drools 5 recently seamlessly integrated powerful Complex Event Processing and workflow capabilities.

²Actually, according to some definitions, production rules are only a specific type of business rules.

2.2.3 Matching algorithms

While modern BRMSs have made tremendous progress in terms of flexibility, user-friendliness and ease-of-use, the underlying implementation techniques arguably have not evolved in pace. Thirty years after its conception by Charles Forgy (1979, 1982), the predominant basis of rule engines today remains the Rete algorithm. In this section, we outline the basic principles of this seminal *matching algorithm*, and compare it with alternative algorithms proposed in research literature.

Matching algorithms

Rule matching algorithms are best explained as if operating using a match-resolve-act cycle. In the *match* phase, the so-called *agenda* is computed. This set contains all applicable (*rule*) *activations*. An activation is a tuple of facts that *match* the LHS of some rule, satisfying all its CEs. Then, a single activation is *selected* from the agenda. This process is called *conflict resolution*, and is determined by a *conflict resolution strategy* (McDermott and Forgy 1978). It is typically governed by heuristics (fact recency, LHS complexity, ...) or execution control mechanisms (e.g. salience). The selected activation is *fired* in the *act* phase, executing all actions of the RHS using the implied variable bindings. This cycle is repeated until the agenda is empty.

Rete

Rete is an *incremental* matching algorithm. It does not recompute the agenda from scratch in each match phase, but instead updates it *incrementally* during the act phase, adding and removing necessary activations after each action. The underlying intuition is that the agenda only changes slightly in each iteration.

As the agenda is typically implemented as a priority queue (Cormen et al. 2009, §6.5), in Rete the distinct match-resolve-act phases are essentially completely amalgamated. The same is true in all realistic matching algorithms.

The name Rete stems from the Latin word for ‘net’, also used in modern Italian to mean ‘network’. To efficiently perform agenda updates, the Rete algorithm maintains a network of nodes, each representing one or more CEs found in the program. We will only consider positive CEs to illustrate the basic principles. While the flexibility of adding different kinds of CEs is one of Rete’s merits, adding e.g. negative CEs already significantly complicates the algorithm. For a more detailed treatment of Rete, the interested reader is referred to (Forgy 1979; Giarratano and Riley 1989; Doorenbos 1995).

All asserted and retracted facts are processed by a tree-structured network. At the root, a fact is first passed to the *alpha network*. The *alpha nodes* form a discrimination network, filtering the fact on its template and slot values. Each alpha node performs a specific test, conditionally sending the fact to one or more

other alpha nodes. The leafs of the alpha network are *alpha memories*, each materialising the set of facts matching a single CE. When a fact reaches an alpha memory it is, depending on the action, either added or removed from it.

The next layer of nodes is called the *beta network*. Each *beta node* (or *join node*) is associated with two inputs: an alpha memory (the '*right*' input) and a *beta memory* (the '*left*' input). A beta memory materialises the output of a beta node, and consists of the set of all fact tuples that match a number of different CEs. These beta tuples thus represent partially matched LHSs. Each beta node combines tuples (their left input) with facts (their right input) to obtain extended tuples, matching one additional CE. For the first layer of beta nodes, the left input is a special beta memory containing only the empty tuple.

The beta network works as follows. Each update to an alpha memory is passed to one or more beta nodes. These are said to be *right-activated*. When a beta node is right activated, it matches the incoming fact with all tuples of its left input. All extended tuples that satisfy the node's matching conditions are passed to a beta memory, or, in case they represent fully matched activations, to the agenda. Similarly, each update to a beta memory *left-activates* beta nodes, which then match the arriving tuple with all facts from their right inputs.

Alternative matching algorithms

The Rete algorithm is characterised by the following properties:

1. Rete is an *eager* matching algorithm, meaning it first computes the entire agenda before selecting the next rule to fire.
2. Rete is an incremental matching algorithm.
3. Rete materialises all partial matches.

The first and third of these properties, while surely beneficial in certain specific cases, mostly lead to poor space and time performance.

Join indexing Because rules often have overlapping LHSs, naive evaluation computes the same (partial) matches many times. By materialising all intermediate matches in beta memories, Rete aims to compute them only once, storing them for later reuse. We call this *join indexing*, after a similar technique in database implementation (see e.g. Valduriez 1987). But, as with all forms of indexing, the time gained by reusing partial matches should be weighed against the inherent cost: if indexed matches are rarely reused, or are frequently retracted again, join indexing only decreases space and time performance. Most studies indeed confirm that Rete's injudicious join indexing strategy mostly works counter-productive (Miranker 1987; Miranker et al. 1990; Miranker and Lofaso 1991; Brant et al. 1991; Wang and Hanson 1992; Obermeyer and Miranker 1994; Wright and Marshall 2003; Van Weert 2010a).

Based on exactly that observation, Miranker et al. (1987, 1991) designed the TREAT algorithm. TREAT is an eager, incremental matching algorithm that *never* performs join indexing. While this means partial matches may be recomputed repeatedly, TREAT was found to mostly outperform Rete (Brant et al. 1991; Wang and Hanson 1992). The reduced memory requirements made TREAT particularly suited for the integration of rules with databases (Hanson 1992; see also Section 2.1 for a brief discussion on active databases).

Of course, never indexing is not always optimal either (Wang and Hanson 1992). Several authors have therefore proposed hybrid approaches, attempting to perform join indexing only in those cases where it is beneficial (Fabret et al. 1993; Hanson et al. 1995; Wright and Marshall 2003). The Gator (‘Generalised TREAT/Rete’) networks proposed by Hanson et al. (1995) additionally allow more general network structures.

Lazy matching Miranker et al. (1990) and Brant (1993) showed that, typically, many if not most activations put on the agenda are never actually fired. Eager matching algorithms therefore spend a considerable amount of time and space computing superfluous partial matches and activations.

To counter this, Miranker and Brant (1990) proposed a *lazy* matching algorithm called LEAPS (Lazy Evaluation Algorithm for Production Systems). A lazy matching algorithm fires each activation as soon as it is computed. Like TREAT, LEAPS moreover never performs join indexing.

LEAPS served as the basis for the Clips++ system (Obermeyer and Miranker 1994) and its successor Venus (Browne et al. 1994). Aside from the high-level reconstruction by Batory et al. (1994), very little has been published on the implementation of LEAPS-based systems, how to optimise the basic algorithm³, or how to extend it towards more expressive language features.

The matching methodology developed for CHR is very similar to LEAPS, but adds numerous novel program analyses and optimisation techniques. These contributions are discussed at length in Part III of this dissertation. A detailed discussion and comparison of the worst-case time and space complexities of the different matching algorithms is given in Section 8.5.

³Except for (Obermeyer et al. 1995), which stresses the importance of fact indexing.

CLIPS/Jess	OPS5	CHR
working memory	working memory	constraint store
fact	working memory element (WME)	constraint
template	class	predicate
slot	attribute	argument
rule	production	rule
left-hand side	left-hand side	head + guard
right-hand side	right-hand side	body
conditional element	condition element	occurrence
assert	make	(add)
retract	remove	(remove)
salience	/	priority
agenda	conflict set	/
activation	instantiation	rule instance

Table 2.1: Rule-based programming terminology

Chapter 3

Logic and Constraint Programming

Logic is the beginning of wisdom, not the end.

— Leonard Nimoy as **Spock**
in *Star Trek VI: The Undiscovered Country* (1991)

Logic programming (LP) and constraint programming (CP) are well-known declarative programming paradigms strongly related to CHR. In this chapter we establish some basic background and terminology we will sometimes rely on in later chapters. Readers familiar with LP and CP can skim this chapter.

3.1 Logic Programming

Logic programming (LP) is, in its broadest sense, the use of mathematical logic for computer programming. Many logic-based formalisms and paradigms fall under this heading: Answer Set Programming (Simons et al. 2002), Inductive Logic Programming (Lavrac and Dzeroski 1994; Muggleton and Raedt 1994), SAT solving (Moskewicz et al. 2001), and many more. Here, we will consider LP in the narrower sense in which it is more commonly understood, namely the use of logic as both a declarative and procedural representation language, as exemplified by the Prolog programming language.

In Section 3.1.1, we will introduce the basic building blocks of (Prolog-like) LP languages. Section 3.1.2 informally outlines the basic operational semantics and evaluation strategy of LP programs. More advanced features of Prolog are briefly discussed in Section 3.1.3.

We refer to (Sterling and Shapiro 1994; Clocksin and Mellish 2003; Bratko 2008) for complete introductions of Prolog, and to (Lloyd 1987; Lifschitz 1996) for comprehensive discussions of LP foundations.

3.1.1 Basics of logic programming

Terms The central data type of most logic programming languages is the (*Herbrand*) *term*. A term is either a (*logical*) *variable*, a *constant*, or *compound term*. A compound term is inductively defined as $f(t_1, \dots, t_n)$, where f is a *function symbol*, n the *arity*, and the arguments t_i terms. Terms thus represent trees in a flattened, text-only form.

Example 3.1 (Terms). The following are examples of terms:

```
X
some_constant
123
f(X, g(A,12,red))
```

In Prolog, variables are the sole syntactical elements whose identifiers are capitalised.

Unification A key operation in logical programming is *unification*. Formally, unification can be defined in terms of *substitutions*.

Definition 3.1 (Substitution). We define a *substitution* $\theta = (X_1 = \tau_1 \wedge \dots \wedge X_n = \tau_n)$ as a conjunction of *bindings* $X_i = \tau_i$, with X_i a variable and $\tau_i \neq X_i$ a term. A substitution is a *variable renaming* if all terms τ_i are variables.

Definition 3.2 (Instantiation). Applying a substitution θ on a logical expression E , denoted $\theta(E)$, entails simultaneously replacing all occurrences of X_i in E by τ_i . The result, $\theta(E)$, is called an *instantiation* of E . After unification, we say that variables X_i are *bound* to terms τ_i .

Definition 3.3 (Unification). A *unifier* of two expressions E and F is a substitution for which $\theta(E) \equiv \theta(F)$, where \equiv denotes *syntactical equality*. A unifier θ is a *most general unifier (mgu)* of E and F if for each unifier σ of E and F , a substitution γ exists for which $\sigma \equiv \gamma \wedge \theta$.

Example 3.2. The mgu of $f(X, g(A, 12, red))$ and $f(h(Y), g(g(Z), Y, red))$ is $X = h(12) \wedge Y = 12 \wedge A = g(Z)$. The terms *black* and *white* though cannot be unified, and neither can $f(X)$ and $g(X)$, or $f(12, X)$ and $f(X, red)$.

If an mgu exists, it is unique (up to variable renaming); otherwise the two expressions cannot be unified. Operationally, the built-in unification predicate of LP languages either simultaneously instantiates its two operands, or *fails* if these terms cannot be unified. Failure is discussed in more detail later.

Listing 3.1 Example Prolog program

```
mother(griet, peter).
father(jos, peter).
father(jos, koen).

parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \== Y.
```

Logical variables Unlike conventional variables in imperative programming languages, *logical variables* may truly represent unknowns, that is, they can be used even if they do not refer to a value. Unification is used to bind variables to other terms. This is considerably different than the common destructive variable assignment operation of imperative languages. Unification is monotonic, that is, terms can only become further instantiated, and variable bindings cannot be undone (except through backtracking: see later). Expressions such as ‘ $X = black$, $X = white$ ’ or ‘ $X = X + 1$ ’¹ will therefore result in failure.

Variables that are not bound to a constant or compound term are called *free* variables. The unification of two free variables is called *aliasing*. A *ground* term does not contain any free variables. The use of non-ground, partially instantiated terms to represent partial knowledge is quintessential to LP.

Predicates A logic program defines a number of logical *predicates*, on top of the *built-in* predicates provided by the LP language. A predicate is denoted as a *functor/arity* pair (predicate and function symbols are called *functors*). An *atom* is a predicate applied to a number of terms.

Example 3.3. Listing 3.1 shows a simple Prolog program, defining four predicates, `mother/2`, `father/2`, `parent/2`, and `sibling/2`. The intended interpretation of, for instance, the atom ‘`mother(X,Y)`’ is that ‘ X is mother of Y ’.

Clauses Predicates are defined using a number of logical formulae. In Prolog, these formulae are rules called *clauses*. A basic Prolog clause is written as `h :- b1, . . . , bn`. Logically, this corresponds to a so-called *Horn clause*, which is an implication of the form $h \leftarrow b_1, \dots, b_n$. Its consequent, called the *head*, is a single positive atom; the antecedent, the *body*, a conjunction of atoms. If the

¹Unifications such as $X = X + 1$, or more generally $X = f(X)$, actually do not result in failure in Prolog systems, but instead in cyclic terms, where X occurs in the term it is bound to. Only when the so-called *occurs check* of the unification algorithm is enabled, these unifications result in failure. For reasons of efficiency, this occurs check is default disabled.

body is the trivial empty conjunction (also denoted `true`), it may be omitted. Such clauses are called *facts*.

Example 3.4. In Listing 3.1, the `mother/2` and `father/2` predicates are defined by a number of facts. Next, the `parent/2` predicate is defined using two clauses. The first declares that ‘if `X` is a mother of `Y`, then `X` is a parent of `Y`’, the second is analogous. Finally, a single clause defines the `sibling/2` predicate. This clause can be read as ‘`X` and `Y` are siblings if (there exists a `Z` such that) `Z` is parent of `X`, `Z` is parent of `Y`, and `X` and `Y` are not equal’.

Note that variables occurring in the head are implicitly universally quantified, whereas variables only occurring in the body are existentially quantified.

3.1.2 Operational semantics

We now informally introduce the standard operational semantics of logic programs, and Prolog programs in particular.

Backwards chaining LP programs are typically evaluated using a *backwards chaining* inference method called *SLD resolution*. Starting with a conjunction of *goals* to resolve—called the *query*—an LP inference engine works backwards from the consequent to the antecedent of clauses until it reaches facts or built-in knowledge to support these goals².

Backwards chaining is fundamentally opposite to *forward chaining*, the inference method used by both production rules (PR; see Chapter 2) and Constraint Handling Rules (CHR; see Chapter 4). Forward chaining inference starts from a set of known facts, moving from the antecedent to the consequent of rules to infer more facts, possibly until a goal is reached. In the case of both PR and CHR, forward chaining is executed exhaustively, without a real concept of a goal. Forward chaining is often said to be a *data-driven* method, in contrast to *goal-driven* backward chaining inference.

The operational reading of a Prolog clause ‘`h :- b1, . . . , bn.`’ is thus “to resolve a goal `h`, it suffices to resolve `b1`, `b2`, . . . , and `bn`.” A clause is applicable if the its head unifies with a goal. In that case, the goal may be replaced with the conjunction in the body. If all goals are resolved, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program. In that case, the query is said to have *succeeded*, and the *solution* is the conjunction of all generated variable bindings.

Example 3.5. Consider the following trivial Prolog predicate:

```
hello(world) :- writeln('Hello, world!').
```

²Technically, an LP inference engine actually attempts to refute the negation of the user’s query, but the resolution process is easier understood without this double negation.

Suppose the user post a query `hello(X)`, with `X` a free logical variable. Then Prolog attempts to unify `hello(X)` with the head of the clause. Clearly this succeeds, and the clause's body is evaluated. As the `writeln/1` built-in simply prints a given term to standard output, the query is fully resolved. As a result of the unification `X` is bound to `world`, so the reported solution is `X = world`.

Search tree A given goal may unify with multiple clause heads. SLD resolution thus implicitly defines a tree, called the *search tree*, of alternative computations. Each node is associated with a goal conjunction. The root corresponds to the initial query. In each node, SLD resolution selects a goal conjunct. We call this the *active goal*. If the active goal is an atom of a user-defined predicate, each clause that unifies with it gives rise to a child node. Built-ins similarly imply one or more child nodes, or none if the built-in fails—for instance a failed unification. A leaf node is a *success* node, if its associated goal conjunction is empty (also denoted `true`). It is a *failure* node if its active goal does not unify with any clause head, or is a failed built-in.

Nondeterminism The abstract SLD resolution mechanism is highly *nondeterministic*. It does not specify which conjunct to select as the active goal, nor in which order the nodes of the search tree are traversed. Prolog implements a more *refined*, deterministic instance of SLD resolution. We discuss this next.

Chronological backtracking Prolog expands goal conjuncts *left-to-right*, in the order in which they appear in the query or clause body, and tries clauses in a textual *top-down* order. In other words: Prolog traverses a naturally defined search tree in a depth-first manner. Each time the Prolog engine moves to a child node, the system creates a *choice-point*. If a failure node is reached, all variable bindings that were made since the most recent choice-point was created are undone, and execution continues with the next alternative of that choice-point. This execution strategy is called *chronological backtracking*.

If all branches end in a failed node, the query is said to *fail*. If a success node is reached, Prolog reports the variable bindings of the solution to the user. In most systems, the user can request for multiple solutions.

Asides from the built-in backtracking search, Prolog's SLD resolution method can be thought of as a generalisation of procedure calls in other languages. When a predicate is called, it is first fully evaluated before control returns to evaluate the remainder of the conjunction. For this, Prolog uses a stack, analogous to the call stack used by conventional languages.

Example 3.6. For the program of Listing 3.1, suppose the user's query is `'sibling(peter, X)'`. Then the initial active fact unifies with the head of the sole `sibling/2` clause. The goal expands to `'parent(Z, peter), parent(Z, Y),`

`peter \== Y`’, and its first conjunct becomes active. Because two clauses are now applicable, a choice-point is created. The first clause is tried first, and the goal becomes `mother(Z, peter), parent(Z, Y), ...`, which in the next step simplifies to `parent(griet, Y), peter \== Y`. This branch eventually fails, because `parent(griet, Y)` only succeeds with `Y = peter` (`\==` denotes (syntactical) disequality). Prolog therefore returns to the aforementioned choice-point, undoing the binding `Z = griet`, and tries the second clause of `parent/2`. As a result of this choice, the only solution `X = koen` is found and returned.

3.1.3 Prolog: programming in logic

The Prolog programming language supports several additional language features not yet discussed. In this section, we briefly introduce e.g. disjunction, negation, and (extra-logical) built-in predicates.

Disjunction

Next to conjunction, Prolog also supports *disjunction* in queries and clause bodies. Extending SLD resolution to disjunction is straightforward: operationally, Prolog simply tries the different disjuncts of a disjunction in textual order through chronological backtracking.

Example 3.7. The `parent/2` predicate can also be declared using the following single Prolog clause:

```
parent(X, Y) :- mother(X, Y) ; father(X, Y).
```

Negation

The built-in predicate `\+/1` provides a form of negation called *negation-as-failure*. A goal `\+ G` succeeds if the goal `G` fails. If `G` has at least one solution, `\+ G` fails. SLD-NF is the natural extension of SLD resolution with negation as failure.

Negation as failure, initially defined simply as an operational construct, is fundamentally different from classical logical negation. It is, for instance, both *non-involutive* (i.e., `\+(\+ G)` is not the same as `G`) and *non-monotonic* (informally: adding facts or clauses to the program may cause a previously successful goal `\+ G` to fail).

The formal semantics of negation-as-failure remained an open issue until Clark (1978) formulated the *completion semantics*. Over the years, alternative semantics of logic programs with negation-as-failure (or other forms of negation) have been studied. Seminal contributions include the *stable model semantics* by Gelfond and Lifschitz (1988), and the *well-founded semantics* by Van Gelder, Ross, and Schlipf (1991).

Metaprogramming

Prolog readily facilitates metaprogramming: Prolog programs are themselves sequences of Prolog terms (`:-/2`, `;/2`, and `,/2` are binary infix operators) that are easily read, inspected, and manipulated using built-in reflection mechanisms. The built-in `'=. . '/2` for instance can be used to construct and deconstruct compound terms, and the `call/1` predicate dynamically executes a given goal.

Example 3.8. Negation as failure could be implemented as follows:

```
\+ G :- call(G), !, fail.
\+ _.
```

Arithmetics

Next to terms, Prolog also supports the data types integer and floating point numbers, as well as all traditional arithmetic operations and functions. The `'is'/2` built-in predicate evaluates its second argument, a ground arithmetic expression, and unifies the result with its first argument. If the expression provided is not fully instantiated, a runtime exception is thrown.

Example 3.9. Suppose the logical variable `Y` is bound to the integer 1, then evaluating `X is Y + 1` is equivalent with the unification `'X = 2'`. If `Y` were a free variable a runtime exception would be thrown. Note the difference with `X = Y + 1`. The latter expression simply unifies `'X'` with the compound term `Y + 1` or `'+(Y, 1)`. Expressions such as `1 + 1 is 1 + 1` and `2 = 1 + 1` therefore both fail, whereas `1 + 1 = 1 + 1` and `2 is 1 + 1` trivially succeed.

Lists

While Prolog lists are simply terms that represent singly linked lists, the language does offer convenient syntactic sugar. The empty list is `[]`, and

$$[X, 2, \text{three}] \equiv [X | [2, \text{three}]] \equiv [X, 2 | [\text{three}]] \equiv [X, 2, \text{three} | []]$$

are all alternative notations for a simple heterogeneous list of three elements. Internally, this list is represented as the term `'.(X, '.'(2, '.'(three, [])))`. Prolog offers several list predicates, such as `member/2`, `append/3` and `length/2`.

All-solution predicates

The `findall(Object, Goal, List)` predicate is a standard Prolog predicate that collects data over all solutions of a goal. More precisely: for each solution of the given `Goal`, an instantiated copy of `Object` is added to a list. The final list is unified with `List`. If `Goal` has no solutions, this will be the empty list.

Two additional all-solution predicates are `bagof/3` and `setof/3`. Both predicates are nondeterministic, and backtrack over possible bindings of the free variables in its `Goal` that also do not occur in `Object`. This behaviour can be overridden using the existential quantifier operator `^`. The result of `setof/3` is a sorted, duplicate-free list.

Example 3.10. Calling `findall(X-Y,parent(X,Y),L)` for the program in Listing 3.1 unifies `L` with the list `[griet-peter,jos-peter,jos-koen]`, and calling `findall(X,parent(X,peter),L)` with `[griet,jos]`. To illustrate `bagof/3`, consider `bagof(X,parent(X,Y),L)`. This expression has two solutions, delivered by means of backtracking: `'Y=peter, L=[griet,jos]'` and `'Y=koen, L=[jos]'`. The equivalent of `findall(X,parent(X,Y),L)` in terms of `bagof/3` would be `bagof(X,Y^parent(X,Y),L)` (the latter fails though if no solutions are found).

General-purpose programming

Prolog is a general purpose programming language, and provides various non-logical predicates to perform e.g. input/output. Such predicates have no logical meaning, and are only useful for the side-effects they exhibit on the system. An example of the use of `writeln/1` was seen in Example 3.5.

3.1.4 Conclusion

While Prolog has roots in formal logic, it is first and foremost an elegant and powerful general-purpose programming language. It has an imperative-like operational semantics, extended with built-in chronological backtracking. Many built-in predicates have no logical semantics, and the semantics of negation-as-absence is also very controversial. The subset of Prolog that only uses true logical predicates is called *pure Prolog* (see e.g. Sterling and Shapiro 1994). Many if not most real-life Prolog programs though are non-pure, and contain considerable amounts of procedural code.

3.2 Constraint Programming

Barták (1999) very aptly describes the concept of a *constraint* as follows:

“Constraints arise in most areas of human endeavour. They formalise the dependencies in physical worlds and their mathematical abstractions naturally and transparently. A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. The constraint thus restricts the possible values that variables can take, it represents partial information about the variables of interest. [...] The important feature of constraints is their declarative manner, i.e., they specify

what relationship must hold without specifying a computational procedure to enforce that relationship. ”

Example 3.11. Some examples of constraints are:

- Given a partially filled-in 9×9 grid of digits from 1 to 9, the objective of a Sudoku puzzle is to complete this grid under the following constraints: all values in each column, each row, and each of the nine 3×3 sub-grids must be mutually distinct.
- The sum of the angles in any triangle equals 180 degrees. The value of each angle in degrees lies in the closed interval $[1, 179]$.
- Given that the domain of the variables X , Y , and Z is \mathbb{N} , the set of natural numbers, the following constraint problem has a unique solution:

$$\left\{ \begin{array}{l} X > Y \\ Y \neq Z \\ X = Z + 1 \\ X \leq 2 \end{array} \right.$$

Complex real-world problems can effectively be modelled in terms of constraints on the variables of a solution, and consequently solved by a constraint solver.

Constraint programming (CP) is often embedded in a *host language*. Traditionally, this host language is a logic programming (LP) language. Today most Prolog implementations include one or more libraries for *constraint logic programming (CLP)*. As seen earlier in this chapter, standard LP essentially only supports a single variable domain (Herbrand terms) and a single constraint (equality, solved via unification). The $\text{CLP}(\mathcal{X})$ framework by Jaffar and Lassez (1987) generalises LP, and extends it with additional variable domains \mathcal{X} , complete with different types of constraints over variables from that domain.

Example 3.12. We illustrate the CLP paradigm using the following clause:

```
constraints(X,Y,Z) :- X > Y, Y \= Z, X = Z + 1, X <= 2.
```

If this clause is called with free variables in plain Prolog, an exception will occur. Using a (hypothetical) $\text{CLP}(\mathbb{N})$ library though, evaluating a constraint entails adding it to a so-called *constraint store*. This results in a failure (in the Prolog sense) if the underlying constraint solver detects an inconsistency. Because constraint solving and consistency checking is computationally expensive, a typical constraint solver only checks so-called *local consistency* when constraints are added. In our example, for instance, even though the constraints already imply a unique solution, the solver probably will not detect this while adding constraints to the store. Yet, it may e.g. derive that X cannot be zero, and that Y and Z are at most one. This process is called *constraint propagation*: possible assignments

are removed from the domains associated with variables. Many propagation algorithms exist, differing both in pruning strength and computational cost. Propagation alone though is mostly insufficient: to efficiently solve any non-trivial problem, propagation must be combined with *search*. In a typical CLP library, a full run of the more powerful propagation and search algorithms is initiated by calling specific predicates (commonly called **labeling**). Propagation and search work in tandem. A naive depth-first backtracking search algorithm creates a choice-point, provisionally assigning a value to some unknown variable, after which propagation is run to further prune variable domains. When a (local) inconsistency is found, backtracking occurs. In reality, a wide variety of more intelligent search algorithms is used.

Numerous constraint programming languages and tools exist, standalone or embedded in some host language (LP, imperative, ...), offering a wide variety of variable domains (integer, rational or real numbers, finite domains, booleans, ...) and constraints (equality, inequalities, arithmetics, global constraints, ...) and using an even wider range of propagation and search techniques. A complete introduction is well outside the scope of this section. For this we refer e.g. to (Van Hentenryck and Saraswat 1997; Marriott and Stuckey 1998; Rossi, Van Beek, and Walsh 2006).

Many CP solvers are implemented in an efficient, low-level language (typically C). While efficient, the resulting systems are not always as flexible and extensible. Early CP systems were based on a so-called “black-box” approach: it was very hard to modify them, or to add e.g. new variable domains or constraints. Over the years, many “glass-box” approaches have been proposed to obtain customisable CP systems; Frühwirth (1998) provides an overview of this. Because these approaches remained restricted and low-level, “no box” extensible constraint solvers have been proposed. It is for this purpose that Frühwirth (1992, 1995, 1998) originally designed the Constraint Handling Rules (CHR) language. In Chapter 4, we will see that CHR has since evolved into a powerful, general purpose programming language, used in a wide range of applications.

Chapter 4

Constraint Handling Rules

*You have to learn the rules of the game.
And then you have to play better than anyone else.*

— **Albert Einstein** (1879–1955)
Swiss-American physicist, philosopher and author

Constraint Handling Rules (CHR) is a declarative, rule-based programming language, incorporating core elements of both production rules and constraint logic programming languages. It was originally designed in 1991 by Frühwirth (1992, 1995, 1998) for the special purpose of adding user-defined constraint solvers to a host language. Over the last decade though, CHR has matured to a powerful and elegant general-purpose language with a wide spectrum of application domains (Sneyers et al. 2010).

This chapter provides a broad overview of the CHR programming language, covering a wide range of theoretical and practical aspects. We start by introducing the CHR language, its syntax and distinguishing characteristics in Section 4.1. Next, Section 4.2 reviews the different formal semantics of CHR, and Section 4.3 surveys important well-studied properties of CHR programs. An overview of some powerful language extensions is given in Section 4.4. Sections 4.5 and 4.6 conclude our journey from theory to practice by surveying the many existing implementations and practical applications of the language.

Complementary introductory texts on CHR can be found for instance in (Frühwirth 1998), (Frühwirth and Abdennadher 2003), and (Frühwirth 2009), or in other Ph.D. dissertations such as those of Duck (2005), Schrijvers (2005), De Koninck (2008), and Sneyers (2008). The most comprehensive survey of recent CHR research and practice, on which several sections of this chapter are based, is written by Sneyers, Van Weert, Schrijvers, and De Koninck (2010).

4.1 Introduction

This section provides a gentle introduction to the CHR programming language. We first lay out the general $\text{CHR}(\mathcal{H})$ framework and its generic syntax in Sections 4.1.1–4.1.2. We then introduce CHR, its semantics and distinguishing characteristics using a number of standard examples (Section 4.1.3). In Section 4.1.3, we very briefly compare with closely related paradigms, such as production rules, term rewriting, and constraint (logic) programming.

4.1.1 $\text{CHR}(\mathcal{H})$

CHR is designed to be a programming language extension, adding user-defined constraints and rules to a given host language \mathcal{H} . We denote the host language in which CHR is embedded between round brackets: i.e. $\text{CHR}(\mathcal{H})$ stands for CHR embedded in host language \mathcal{H} . The traditional host languages for CHR are CLP languages, and until recently most systems were $\text{CHR}(\text{Prolog})$ systems. In Section 4.5, however, we will see that several CHR implementations exist for functional and imperative host languages as well. In Chapter 7 we further detail how to effectively embed CHR in imperative host languages.

In the $\text{CHR}(\mathcal{H})$ framework, two types of constraints are distinguished: *built-in constraints* and *CHR constraints*. The former are provided by the host language, as are the data types used by $\text{CHR}(\mathcal{H})$. Prolog’s single data type, for instance, is a term, and in its pure form it offers only one built-in constraint, namely Herbrand term equality, solved by the built-in unification mechanism. CHR programs themselves add CHR constraints to \mathcal{H} . Both the formal, logical semantics of these constraints, and the way they are evaluated by the CHR execution engine, are fully determined by the rules of the CHR program.

Typically the host language is required to at least provide the trivial built-in constraints `true` and `false`, as well as a syntactic equality constraint over its data types.¹ This equality constraint is required for pattern matching.

4.1.2 Syntax

This section introduces the generic syntax of $\text{CHR}(\mathcal{H})$ languages. Specific implementations typically further refine this syntax, tailored towards an optimal integration with the host language. In Chapter 7, for instance, we thoroughly discuss the design of CHR embeddings in imperative host languages.

By default, CHR constraint symbols are drawn from the set of predicate symbols, and denoted by a functor/arity pair. CHR constraints are atoms constructed from these symbols and the data types provided by the host language \mathcal{H} . Most systems offer syntax to declare CHR constraints, often with type and

¹More precisely: only the entailment check (‘ask’ version) of equality is required.

mode declarations for their arguments, as well as alternative ways of writing constraints. For now, these language design issues are abstracted away.

A CHR program is also called a *CHR handler*. It consists of a sequence of *CHR rules*. There are three kinds of rules ($n, n_g, n_b \geq 1$ and $n \geq r > 1$):

- Simplification rules: $h_1, \dots, h_n \Leftrightarrow g_1, \dots, g_{n_g} \mid b_1, \dots, b_{n_b}$.
- Propagation rules: $h_1, \dots, h_n \Rightarrow g_1, \dots, g_{n_g} \mid b_1, \dots, b_{n_b}$.
- Simpagation rules: $h_1, \dots, h_{r-1} \setminus h_r, \dots, h_n \Leftrightarrow g_1, \dots, g_{n_g} \mid b_1, \dots, b_{n_b}$.

The *head* ‘ h_1, \dots, h_n ’ of a rule is a sequence, or conjunction, of CHR constraints. A rule with n head constraints is called an n -headed rule. We also use the term *single-headed* rule if $n = 1$, or *multi-headed* rule if $n > 1$. The conjuncts h_i of the head are called *occurrences*. Both the occurrences in a simplification rule and the occurrences ‘ h_r, \dots, h_n ’ in a simpagation rule are called *removed occurrences*. All other occurrences are *kept occurrences*. The *body* of a CHR rule is a conjunction of CHR constraints and built-in constraints ‘ b_1, \dots, b_{n_b} ’. The part of the rule between the arrow and the body is called the *guard*. It is a conjunction of built-in constraints. The guard ‘ $g_1, \dots, g_{n_g} \mid$ ’ is optional; if omitted, it is considered to be ‘**true** \mid ’. A rule is optionally preceded by a unique *rule identifier*, followed by the ‘@’ symbol. Rules without an explicit identifier commonly get one implicitly.

4.1.3 CHR by example

This section illustrates some of CHR’s unique characteristics, and explains informally the meaning of CHR rules, and how a CHR program operates. Operationally, CHR rules behave much like production rules (cf. Chapter 2).

The CHR runtime operates on a database-like *multiset* of CHR constraints called the (*CHR*) *constraint store*. This is similar to the working memory of a production rule system. The fact though that the store is a multiset—i.e. that multiple instances of syntactically equal CHR constraints can co-exist—is a distinguishing characteristic of CHR.

A CHR execution or *derivation* starts from an initial *query* given by the user. The derivation proceeds by applying, or *firing*, rules of the program. When no more rules can be applied, the derivation ends; the final store is called the *solution* or *solved form*. CHR rules are thus executed using forward chaining, much like production rules, but contrary to e.g. Prolog: see Section 3.1.2.

A rule is *applicable* if the current constraint store contains CHR constraints that *match* the rule’s occurrences, and for which the guard condition hold. To determine applicability, CHR uses pattern *matching* rather than unification. Formally, a conjunction of CHR constraints C matches a rule’s head H if a (*matching*) *substitution* θ exists for which $C \equiv \theta(H)$. As seen in Section 3.1, unification would entail the existence of a unifier η for which $\eta(C) \equiv \eta(H)$.

Listing 4.1 The CHR(Prolog) program LEQ, a solver for less-than-or-equal constraints.

```

reflexivity @ leq(X,X) ⇔ true.
antisymmetry @ leq(X,Y), leq(Y,X) ⇔ X = Y.
idempotence @ leq(X,Y) \ leq(X,Y) ⇔ true.
transitivity @ leq(X,Y), leq(Y,Z) ⇒ leq(X,Z).

```

Listing 4.2 The linearised form of the LEQ handler.

```

reflexivity @ leq(X,X1) ⇔ X1 = X | true.
antisymmetry @ leq(X,Y), leq(Y1,X1) ⇔ X1 = X, Y1 = Y | X = Y.
idempotence @ leq(X,Y) \ leq(X1,Y1) ⇔ X1 = X, Y1 = Y | true.
transitivity @ leq(X,Y), leq(Y1,Z) ⇒ Y1 = Y | leq(X,Z).

```

Matching is therefore also referred to as *one-way unification*. To illustrate the difference, compare the following CHR(Prolog) example to Example 3.5:

Example 4.1 (Matching). Suppose the store contains a CHR constraint $c(X)$ with X a free logical variable, and the program contains a rule:

```
c(world) <=> writeln('Hello, world!').
```

The $c(X)$ constraint does not match with the rule's head. Matching is only allowed to instantiate variables occurring in the rule's head, and not those occurring in CHR constraints in the store. Conversely, the following rule is applicable for any $c/1$ constraint, i.e. $c(\text{world})$, $c(\text{peter})$, $c(Y)$, etc.:

```
c(X) <=> writef('Hello, %t!\n', [X]).
```

This example also shows that, in practice, CHR typically also allows \mathcal{H} host-language statements (functions, predicates, ...) that are not strictly speaking constraints. In the formal CHR(\mathcal{H}) framework, these are mostly still modelled as 'built-in constraints' (or assumed not to be used).

Example 4.2 (Non-linear patterns). CHR allows so-called *non-linear patterns*, that is, the same variable may occur more than once in the head, possibly in more than one conjunct. This is for instance the case in the LEQ handler in Listing 4.1, a classic CHR(Prolog) program to solve less-than-or-equal constraints (its operational semantics is demonstrated below in Example 4.3).

For a rule to be applicable, its guard must of course also be satisfied. One way to understand the non-linear matching mechanism is to view them as implicit equality guards. In the *linearised form* or *Head Normal Form (HNF)* of a rule,

all such guards are made explicit. Listing 4.2 contains the linearised form of the LEQ program. We will sometimes use \mathcal{P}^* to denote the normalised (linearised) version of a CHR program \mathcal{P} .

Rules modify the constraint store in the following way. A simplification rule can be considered as a rewrite rule which replaces the left- with the right-hand side, provided the guard holds. The double arrow indicates that the head is logically equivalent to the body, which justifies the replacement. The intention is that the body is a simpler or more canonical form of the head. In propagation rules, the body is a consequence of the head: given the head, the body may be added (if the guard holds). Logically, the body is implied by the head so it is redundant. However, adding redundant constraints may allow simplifications later on. Simplification rules are a hybrid between simplification and propagation rules: the constraints before the backslash are kept, while the constraints after the backslash are removed.

Example 4.3 (Rule application). Reconsider the LEQ CHR(Prolog) program in Listing 4.1. The first rule, *reflexivity*, replaces trivial constraints $\text{leq}(X, X)$ with **true**. Operationally, this entails removing such constraints from the constraint store. The second rule, *antisymmetry*, states that $\text{leq}(X, Y)$ and $\text{leq}(Y, X)$ are logically equivalent to $X = Y$. Operationally, this means that constraints matching the left-hand side may be removed from the store, after which the Prolog built-in equality constraint solver is used to unify X and Y . The third rule, *idempotence*, removes redundant copies of the same $\text{leq}/2$ constraint. It is necessary to do this explicitly since CHR has *multiset semantics*. The last rule, *transitivity*, is a propagation rule that computes the transitive closure of the $\text{leq}/2$ relation.

An example derivation could be as follows:

$$\begin{array}{lcl}
 & & \text{leq}(A,B), \text{leq}(B,C), \text{leq}(C,A) \\
 (\textit{transitivity}) & \rightarrow & \underline{\text{leq}(A,B), \text{leq}(B,C), \text{leq}(C,A)}, \underline{\text{leq}(B,A)} \\
 (\textit{antisymmetry}) & \rightarrow & \underline{\text{leq}(B,C), \text{leq}(C,A)}, A = B \\
 (\textit{Prolog}) & \rightarrow & \underline{\text{leq}(A,C), \text{leq}(C,A)}, A = B \\
 (\textit{antisymmetry}) & \rightarrow & \underline{A = C}, \underline{A = B}
 \end{array}$$

Without the constraints initially propagated by the *transitivity* rule, no constraint simplification would have been possible. This illustrates the power of complementing rewrite rules with propagation rules. The example also demonstrates the interaction with the host language through built-in constraints: solving the equality constraint in Prolog triggers the *antisymmetry* rule.

Example 4.4 (General purpose programming). Listing 4.3 lists another simple CHR program called PRIMES, a CHR variant of the Sieve of Eratosthenes. Dating back to at least (Frühwirth 1992), this is one of the first examples where CHR is used more as a general-purpose programming language. Given a query of $\text{upto}(n)$

Listing 4.3 The CHR program PRIMES, a prime number sieve.

```

generate @ upto(N) ⇔ N > 1 | prime(N), upto(N-1).
done     @ upto(1) ⇔ true.
sieve   @ prime(A) \ prime(B) ⇔ B mod A = 0 | true.

```

with n a natural number, it computes all prime numbers up to n . Provided $N > 1$, the first rule (*generate*) ‘simplifies’ `upto(N)` to `upto(N-1)` and adds a `prime(N)` constraint. The second rule handles the case for $N = 1$, removing the `upto(1)` constraint (by simplifying it to the built-in constraint `true`). The third and most interesting rule (*sieve*) is a simpagation rule. If there are two `prime/1` constraints `prime(A)` and `prime(B)`, such that B is a multiple of A , the latter constraint is removed. The effect of this rule is that non-primes are removed. After exhaustively applying the rules, the solution corresponds exactly to the prime numbers up to n .

If more than one (simplifying) rule is applicable, CHR chooses one applicable rule. As discussed in detail in Section 4.2, depending on the operational semantics used, this choice is either nondeterministic (possibly restricted by e.g. priorities), based on rule order (similar to Prolog’s refined SLD algorithm).

In any case, CHR is said to *commit* to these choices. In other words, unlike in Prolog, alternative options are no longer considered. This is called *committed choice* or *don’t care nondeterminism*, in contrast with the *don’t know nondeterminism* used by LP languages (Section 3.1.2).

Example 4.5 (Committed choice). Consider the following simple Prolog (to the left) and CHR programs (to the right):

<code>throw(Coin) :- Coin = head.</code>	<code>throw(Coin) ⇔ Coin = head.</code>
<code>throw(Coin) :- Coin = tail.</code>	<code>throw(Coin) ⇔ Coin = tail.</code>

In Prolog, for a goal `coin(X)`, the first clause is tried first. If this leads to a failure, or more solutions are required, the runtime will backtrack over this choice, and try the second clause. In CHR on the other hand, if one of the rules removes some `coin(X)` constraint, the other rule is never tried, even if the chosen rule leads to failure.

4.1.4 Relation to other formalisms

CHR can be seen as a restricted production rule language (Chapter 2), augmented with powerful elements from CLP languages (Chapter 3). A comparison of some key characteristics is given in Table 4.1.

We now briefly discuss the relation of CHR to other well-known formalisms. A more elaborate comparison with these and other related formalism can be found (Sneyers et al. 2010, Section 6).

	CHR	PR	CLP
Basic element	constraint	fact / WME	predicate
Rules	rule	production	clause
Multi-headed	✓	✓	
Rule applicability	matching	matching	unification
Non-ground data	✓ ^a		✓
Constraints	✓ ^a		✓
Chaining	forward	forward	backward
Nondeterminism	don't care ^b	don't care	don't know

Table 4.1: Comparison of CHR with production rules (PR) and Prolog-like CLP languages.

^aMost CHR systems, even for non-CLP host languages (see Chapter 7), support at least logical variables and simple built-in constraints (e.g. equality).

^bExtensions of CHR with don't know nondeterminism and search are discussed in Section 4.4.4.

Term rewriting CHR can be considered as associative and commutative (AC) term rewriting, restricted to flat conjunctions, but augmented with propagation and constraint-logical language features. The term rewriting literature inspired many results for CHR, most notably on confluence and termination (see Section 4.3). Recently, Duck, Stuckey, and Brand (2006) proposed the ACD term rewriting, which subsumes both AC term rewriting and CHR.

Join calculus The join-calculus is a calculus for concurrent programming that inspired languages such as JoCaml, Join Java, and Polyphonic C#. Join-calculus rules, or chords, are essentially guardless simplification rules with linear match patterns. Sulzmann and Lam (2007b, 2008) clearly show CHR's superior expressiveness (propagation, general guards, non-linear patterns, ...).

Petri nets Petri nets are another well-known formalism for modelling and analysing concurrent processes. Betz (2007) provides a first study of the relation between CHR and Petri nets. CHR is more expressive than standard place/transition nets (P/T nets), that, unlike CHR, are not Turing complete. Betz (2007) also presents a translation of a significant subsegment of CHR into coloured Petri nets, a more expressive, Turing complete extension of Petri net.

4.2 Formal Semantics

The semantics of CHR is defined either as a *logical semantics* (Section 4.2.1), or as an *operational semantics* (Sections 4.2.2–4.2.3). The former formally maps programs to logical theories, thus forming the formal foundations of the CHR

language, whereas the latter models the runtime behaviour of programs. As CHR rules are essentially executable logical specifications, relatively strong soundness and completeness results have been proven that directly link the operational semantics to the underlying logical semantics.

4.2.1 Logical semantics

Classical Logic Semantics

The (classical) logical semantics (Frühwirth 1998) of a CHR program—also called its *logical reading* or *declarative semantics*—is defined as the conjunction of the logical formulas for each rule, with the built-in constraint theory $\mathcal{D}_{\mathcal{H}}$. The latter determines the semantics of the built-in constraints of the host language \mathcal{H} ; for a rigorous definition, we refer e.g. to (Schrijvers 2005).

Let \bar{x} denote the variables occurring only in the body of the rule. We use $\bar{\forall}(F)$ to denote universal quantification over all free variables in F . A simplification rule $H \Leftrightarrow G \mid B$ corresponds to a logical equivalence, under the condition that the guard is satisfied: $\bar{\forall}(G \rightarrow (H \Leftrightarrow \exists \bar{x}B))$. Similarly, a propagation rule $H \Rightarrow G \mid B$ corresponds to a conditional logical implication $\bar{\forall}(G \rightarrow (H \rightarrow \exists \bar{x}B))$, and a simpagation rule $H_k \setminus H_r \Leftrightarrow G \mid B$ to a conditional equivalence: $\bar{\forall}(G \rightarrow (H_k \rightarrow (H_r \Leftrightarrow \exists \bar{x}B)))$.

Example 4.6. As an example, consider the program LEQ of Listing 4.1. The logical formulas corresponding to its rules are:

$$\left\{ \begin{array}{ll} \forall x, y : x = y \rightarrow (\text{leq}(x, y) \Leftrightarrow \text{true}) & (\text{reflex.}) \\ \forall x, y, x', y' : x = x' \wedge y = y' \rightarrow (\text{leq}(x, y) \wedge \text{leq}(y', x') \Leftrightarrow x = y) & (\text{antisym.}) \\ \forall x, y, x', y' : x = x' \wedge y = y' \rightarrow (\text{leq}(x, y) \rightarrow (\text{leq}(x', y') \Leftrightarrow \text{true})) & (\text{idemp.}) \\ \forall x, y, y', z : y = y' \rightarrow (\text{leq}(x, y) \wedge \text{leq}(y', z) \rightarrow \text{leq}(x, z)) & (\text{trans.}) \end{array} \right.$$

or equivalently:

$$\left\{ \begin{array}{ll} \forall x : \text{leq}(x, x) & (\text{reflexivity}) \\ \forall x, y : \text{leq}(x, y) \wedge \text{leq}(y, x) \Leftrightarrow x = y & (\text{antisymmetry}) \\ \text{true} & (\text{idempotence}) \\ \forall x, y, z : \text{leq}(x, y) \wedge \text{leq}(y, z) \rightarrow \text{leq}(x, z) & (\text{transitivity}) \end{array} \right.$$

Note the strong correspondence between the syntax of the CHR rules, their logical reading, and the natural definition of partial order. CHR's multiset semantics, however, is lost in its classical logical reading, as the *idempotence* rule corresponds to a logical tautology.

Not all programs, however, have a meaningful classical logical semantics.

Example 4.7. The rules of the PRIMES program of Listing 4.3 have no meaningful logical reading. The semantics of the *sieve* rule, e.g., is logically equivalent to:

$$\forall a, b : \text{prime}(a) \wedge a|b \rightarrow \text{prime}(b)$$

This formula nonsensically states a number is prime if it has a prime factor.

Example 4.8. Another distinctive feature of CHR’s operational semantics that cannot be modelled in classical logic is committed choice. The logical reading of the simple COIN program from Example 4.5 for instance is:

$$\forall x, y : (\text{throw}(x) \leftrightarrow x = \text{head}) \wedge (\text{throw}(y) \leftrightarrow y = \text{tail})$$

which erroneously implies that `head = tail`.

Linear Logic Semantics

For general-purpose CHR programs such as PRIMES, or programs that rely on CHR’s multiset semantics, the classical logic reading is often inconsistent with the intended meaning (see previous section). To overcome these limitations, Bouissou (2004) and Betz and Frühwirth (2005, 2007) independently proposed an alternative declarative semantics based on (intuitionistic) linear logic (Girard 1987). By interpreting CHR constraints as linear resources, and CHR rules as linear implication rather than logical equivalence, Betz and Frühwirth (2005, 2007) obtain surprisingly strong soundness and completeness results.

Example 4.9. The linear logic reading of the *sieve* rules of the PRIMES program (Listing 4.3) is:

$$!\forall(a|b \multimap (\text{prime}(b) \otimes \text{prime}(a) \multimap \text{prime}(a)))$$

which adequately models the intended behaviour of the rule (see Betz and Frühwirth (2005) for a proper explanation).

Example 4.10. The linear logic reading of the coin-throwing program of Example 4.5 is logically equivalent to:

$$!\forall(\text{throw}(\text{Coin}) \multimap (\text{Coin} = \text{head}) \& (\text{Coin} = \text{tail}))$$

In natural language, this formula means “you can always replace `throw(Coin)` with either `Coin = head` or `Coin = tail`, but not both”. This corresponds to CHR’s committed-choice and unidirectional forward chaining rule application.

Transaction Logic Semantics

The linear logic semantics is closer to the operational semantics than the CHR classical logical semantics. However, it still does not allow precise reasoning about CHR derivations: while derivations correspond to proofs of logic equivalence of the initial and the final state, it only allows reasoning on the result of an execution, not on the execution itself. The transaction logic semantics (Meister, Djelloul, and Robin 2007) aims to bridge this remaining gap between the logical and operational semantics of CHR.

4.2.2 The theoretical operational semantics ω_t

The operational semantics of CHR describes the runtime behaviour of the language’s implementations. The most general operational semantics is the so-called *high-level* or *theoretical operational semantics* (Frühwirth 1998), commonly denoted ω_t . It models all standard CHR operational characteristics, such as its multiset store, matching, forward chaining rule firing, and reapplication prevention. Nearly all CHR systems implement this semantics.

The ω_t semantics is formulated as a state transition system, where the relation between consecutive execution states is defined by transition rules.

Execution states

Definition 4.1 (Identified constraints). An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with a unique *constraint identifier* i . This integer number serves to differentiate between copies of the same constraint. We further introduce projection operators $\text{CHR}(c\#i) = c$ and $\text{ID}(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner—e.g., $\text{ID}(S) = \{i \mid c\#i \in S\}$.

Definition 4.2. An *execution state* σ is a tuple $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$, where:

- The *goal* \mathbb{G} is a multiset of constraints to be rewritten to solved form.
- The CHR *constraint store* \mathbb{S} is a set of identified CHR constraints that can be matched with rules in the program \mathcal{P} . Note that due to the constraint identifiers, \mathbb{S} is a set, even though $\text{CHR}(\mathbb{S})$ is a multiset.
- The *built-in constraint store* \mathbb{B} is an abstract conjunction representing all built-in constraints that have been posted to the underlying solver. These constraints are assumed to be solved (implicitly) by the *built-in constraint solver(s)* offered by the host language \mathcal{H} .
- The *propagation history* \mathbb{T} is a set of tuples, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. The propagation history is used to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire with the same set of constraints only once.²
- Finally, the counter $n \in \mathbb{N}$ represents the next integer that can be used to number a CHR constraint.

²Early work on CHR, as well as some more recent publications (e.g., Bouissou 2004; Duck et al. 2007; Haemmerlé and Fages 2007), use a *token store* instead of a propagation history (this explains the convention of denoting the propagation history with \mathbb{T}). A token store contains a token for every potential (future) rule application, which is removed when the rule is actually applied. The propagation history formulation is dual, but closer to most implementations. Confusingly, the term *token store* has also been used for what is commonly referred to as the “propagation history” (e.g., Chin et al. 2003; Tacchella et al. 2007).

1. **Solve.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$
where c is a built-in constraint and $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset} \mathbb{B}$.
2. **Introduce.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
where c is a CHR constraint and $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset} \mathbb{B}$.
3. **Apply.** $\langle \mathbb{G}, K \sqcup R \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \uplus \mathbb{G}, K \sqcup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \sqcup \{t\} \rangle_n$
where $\rho @ H_k \setminus H_r \Leftrightarrow G \mid B$ is a renamed apart rule of \mathcal{P} ,
 $\theta \in \text{matchings}(H_k, H_r, G, K, R, \mathbb{B})$, and $t = (\rho, \text{ID}(K) ++ \text{ID}(R)) \notin \mathbb{T}$.

Figure 4.1: The transition rules of the theoretical operational semantics ω_t , defining $\mapsto_{\mathcal{P}}$. We use \uplus for multiset union. For constraint conjunctions B_1 and B_2 , $\bar{\exists}_{B_2}(B_1)$ denotes $\exists X_1, \dots, X_n : B_1$, with $\{X_1, \dots, X_n\} = \text{vars}(B_1) \setminus \text{vars}(B_2)$.

We use $\sigma, \sigma_0, \sigma_1, \dots$ to denote execution states, and Σ^{CHR} to denote the set of all execution states.

Transition rules

For a given CHR program \mathcal{P} , the transitions are defined by the binary relation $\mapsto_{\mathcal{P}} \subset \Sigma^{\text{CHR}} \times \Sigma^{\text{CHR}}$ shown in Figure 4.1. The first two transitions add constraints from the goal to either the built-in and CHR constraint store. The most interesting transition is **Apply**, which fires a rule. Formally, matching substitutions and rule applicability (cf. Section 4.1.3) are defined as follows:

Definition 4.3 (Matching substitutions). Let H_k and H_r be sequences of CHR constraints, G and \mathbb{B} conjunctions of built-in constraints, and K and R sequences of identified CHR constraints. Then the set of matchings substitutions $\text{matchings}(H_k, H_r, G, K, R, \mathbb{B})$ is defined as those substitutions for which $\theta(H_k) = \text{CHR}(K)$, $\theta(H_r) = \text{CHR}(R)$, and $\mathcal{D}_{\mathcal{H}} \models (\bar{\exists}_{\emptyset} \mathbb{B}) \wedge (\mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G))$.

Using a matching substitution θ , a *rule instance* $\theta(\rho)$ instantiates a rule ρ with CHR constraints matching its head. The propagation history prevents trivial non-termination by ensuring that each rule instance fires at most once (in particular if ρ is a propagation rule).

Derivations

Execution proceeds by exhaustively applying the transition rules, starting from an initial state.

Definition 4.4 (Initial state). Given a user-defined *query* Q , a multiset of CHR and built-in constraints, an *initial execution state* is given by $\langle Q, \emptyset, \text{true}, \emptyset \rangle_1$. The set of all initial states is denoted Σ^{init} .

Definition 4.5 (Final state). A final state $\omega_f = \langle \emptyset, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ is an execution state for which no transition applies. We distinguish two types of final states. In a *failure state*, the built-in constraint store is inconsistent, i.e., $\mathcal{D}_{\mathcal{H}} \models \neg \exists_0 \mathbb{B}$. All other final states are *successful final states*. The set of final states is Σ^{final} .

Definition 4.6 (Derivation). A derivation D is a (possibly infinite) sequence of execution states, with $D[1] \in \Sigma^{init}$, and $D[i] \xrightarrow{\mathcal{P}} D[i+1]$ for $1 \leq i < |D|$. For finite derivations $D[|D|] \in \Sigma^{final}$. We also use the notational abbreviation $\sigma_1 \xrightarrow{*}_{\mathcal{P}} \sigma_n$ to denote a finite derivation $[\sigma_1, \dots, \sigma_n]$.

Definition 4.7 (Equivalence). All failed states are *equivalent*. Two successful final states $\sigma = \langle \emptyset, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ and $\sigma' = \langle \emptyset, \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_{n'}$ are equivalent iff a variable renaming θ exists such that $\text{CHR}(\mathbb{S}) = \text{CHR}(\theta(\mathbb{S}'))$ and $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \leftrightarrow \theta(\mathbb{B}')$.

Soundness and completeness

Frühwirth (1998) surveys the relatively strong soundness and completeness results that link the ω_t semantics and the classical logical semantics, particularly for terminating and confluent (also called: *well-behaved*) CHR programs. As discussed in 4.2.1, Betz and Frühwirth (2005, 2007) obtained even stronger results when considering linear logic instead.

4.2.3 The refined operational semantics ω_r

The ω_t semantics is highly nondeterministic: it does not determine the order in which constraints are **Solved** and **Introduced**, nor does it determine the order in which the rules are fired by **Apply** transitions. The actual runtime behaviour of concrete implementations, however, is mostly far more deterministic. They use specific matching and evaluation algorithms, under which typically only a limited subset of all ω_t derivations is possible.

As explained in more detail in Section 4.5, most current CHR systems are based on the implementation strategy developed by Holzbaur and Frühwirth (2000a). To better model the behaviour of these systems, Duck et al. (2004) defined the so-called refined operational semantics ω_r .

The ω_r refinement of ω_t is completely analogous to Prolog's refinement of the abstract SLD resolution mechanism (cf. Section 3.1.2). That is: CHR rules are tried in a top-to-bottom textual order, and both queries and rule bodies are executed from left to right, treating each constraint conjunct like a traditional procedure call. A central concept is the *active constraint*. Similar to the active goal in Prolog, the active constraint traverses all its occurrences in a textual order, searching for matching rule instances.

Example 4.11. Listing 4.4 shows the *occurrence numbers* of the PRIMES handler. Occurrences are numbered, per constraint, in the order in which they are matched by ω_r (i.e. in a top-down, right-to-left manner).

Listing 4.4 Occurrence numbering for the PRIMES handler: occurrence numbers are shown using ^[j] annotations.

```

generate @ upto(N)[1] ⇔ N > 1 | prime(N), upto(N-1).
done     @ upto(1)[2] ⇔ true.
sieve   @ prime(A)[2] \ prime(B)[1] ⇔ B mod A = 0 | true.

```

We now formally define ω_r as a state-transition system (Duck et al. 2004). This definition is similar to that of ω_t in the previous section.

Execution states

Definition 4.8 (Occurred constraints). An *occurred* constraint $c\#i:j$ is an identified constraint $c\#i$ annotated with an occurrence number j . This annotation indicates that only matches with the j 'th occurrence of c will be considered when $c\#i:j$ is active.

Definition 4.9 (Execution state). A state in ω_r is a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$, with \mathbb{S} , \mathbb{B} , and \mathbb{T} defined as in Definition 4.2. The *activation stack* \mathbb{A} is a heterogeneous sequence of constraints, identified constraints, and occurred constraints.

Initial and final states are defined as before. If the top-most element of \mathbb{A} is an occurred constraint, it is called the *active constraint*. Other occurred constraints in \mathbb{A} are called *suspended constraints*.

Transition rules

Figure 4.2 lists the transition rules of ω_r . If a rule is fired in a **Propagate** or **Simplify** transition, the body is pushed onto the execution stack \mathbb{A} , thus *suspending* the previously active constraint. A suspended constraint becomes active again once all body conjuncts have been activated (for CHR constraints) or solved (for built-in constraints) in a left-to-right order. In other words, \mathbb{A} acts as a standard call stack, and is used to execute constraints much like procedure calls are executed in traditional stack-based programming languages.

If the top of the stack is an unidentified CHR constraint c , the **Activate** transition adds c to the constraint store, and then *activates* it (i.e. adds an occurrence number). This causes all the constraint's occurrences to be tried in order. When an occurred identified CHR constraint $c\#i:j$ is active, only matches with the j 'th occurrence of c 's constraint are considered. Interleaving a sequence of **Default** transitions, all applicable rules are fired in **Propagate** and **Simplify** transitions. A constraint remains active, if not temporarily suspended, until it is removed by a **Simplify** transition, or until all its occurrences have been traversed and a **Drop** transition is applied.

1. **Solve** $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle S \text{ ++ } \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if b is a built-in constraint.
For the set of *reactivated constraints* $S \subseteq \mathbb{S}$, the following bounds hold:
 - Lower bound: $\forall S' \subseteq \mathbb{S} : (\exists \rho \in \mathcal{P} : \neg \text{applicable}(\rho, S', \mathbb{B}) \wedge \text{applicable}(\rho, S', b \wedge \mathbb{B})) \rightarrow (S' \cap S \neq \emptyset)$
 - Upper bound: $\forall c \in S : \text{vars}(c) \not\subseteq \text{fixed}(\mathbb{B})$
2. **Activate** $\langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle [c\#n : 1|\mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$ if c is a CHR constraint (which has not yet been active or stored in \mathbb{S}).
3. **Reactivate** $\langle [c\#i|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle [c\#i : 1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ if c is a CHR constraint (re-added to \mathbb{A} by a **Solve** transition but not yet active).
4. **Simplify** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \text{ ++ } \mathbb{A}, K \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T} \sqcup \{t\} \rangle_n$ with $\mathbb{S} = K \sqcup R_1 \sqcup \{c\#i\} \sqcup R_2 \sqcup S$, if the following holds:
 - The j -th occurrence of c in \mathcal{P} is $H_r[k]$ in a (renamed apart) rule $\rho @ H_k \setminus H_r \Leftrightarrow G \mid B$
 - $|R_1| = k - 1$
 - $\theta \in \text{matchings}(H_k, H_r, K, R_1 \text{ ++ } [c\#i] \text{ ++ } R_2, \mathbb{B})$
 - $t = (\rho, \text{ID}(K \text{ ++ } R_1) \text{ ++ } [i] \text{ ++ } \text{ID}(R_2)) \notin \mathbb{T}$
5. **Propagate** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \text{ ++ } [c\#i : j|\mathbb{A}], \mathbb{S} \setminus R, \theta \wedge \mathbb{B}, \mathbb{T} \sqcup \{t\} \rangle_n$ with $\mathbb{S} = K_1 \sqcup \{c\#i\} \sqcup K_2 \sqcup R \sqcup S$, if the following holds:
 - The j -th occurrence of c in \mathcal{P} is $H_k[k]$ in a (renamed apart) rule $\rho @ H_k \setminus H_r \Leftrightarrow G \mid B$
 - $|K_1| = k - 1$
 - $\theta \in \text{matchings}(H_k, H_r, K_1 \text{ ++ } [c\#i] \text{ ++ } K_2, R, \mathbb{B})$
 - $t = (\rho, \text{ID}(K_1) \text{ ++ } [i] \text{ ++ } \text{ID}(K_2 \text{ ++ } R)) \notin \mathbb{T}$
6. **Drop** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ if c has no j -th occurrence in \mathcal{P} .
7. **Default** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle [c\#i : j + 1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ if the current state cannot fire any other transition.

Figure 4.2: The transition rules of the refined operational semantics ω_r . All transitions have as an additional, implicit precondition that $\mathcal{D}_{\mathcal{H}} \models \exists_0 \mathbb{B}$.

If the top-most element of \mathbb{A} is a built-in constraint, it is passed to the built-in solver in a **Solve** transition. As this may affect the entailment of guards, CHR constraints must be *reactivated* to ensure all newly applicable rule instances are found. **Solve** therefore re-adds a set of *reactivated constraints* to the stack, which then become active again, one by one, by a series of **Reactivate** transitions.

The set of reactivated constraints is not fully determined. Instead, reasonable lower and upper bounds are specified³. The lower bound ensures that, for each previously inapplicable rule instance, at least one of the constraints matching this instance is reactivated. The upper bound specified in the **Solve** transition prohibits so-called *fixed constraints* from being reactivated. If all constraint arguments are already completely fixed before, adding a new built-in constraint cannot enable additional rule instances. Formally:

Definition 4.10. A variable v is *fixed* by constraint conjunction \mathbb{B} , denoted $v \in \text{fixed}(\mathbb{B})$, iff $\mathcal{D}_{\mathcal{H}} \models \forall \theta \left((\pi_{\{v\}}(\mathbb{B}) \wedge \pi_{\{\theta(v)\}}(\theta(\mathbb{B}))) \rightarrow v = \theta(v) \right)$ for any variable renaming θ .

Correspondence The refined operational semantics is an instance of the abstract ω_t semantics. That is, all ω_r derivations are also valid ω_t derivations.

Theorem 4.1. *For all derivations D under ω_r , there exists a corresponding derivation D' under ω_t .*

Proof. We define the straightforward abstraction function α , that maps states and derivations of ω_r as follows:

$$\begin{aligned} \alpha(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) &= \langle \{c \in \mathbb{A} \mid c \text{ is not of form } c\#i \text{ or } c\#i:j\}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \\ \alpha(\square) &= \square \\ \alpha([\sigma_1|D]) &= \begin{cases} \alpha(D) & \text{if } |D| > 0 \text{ and } \alpha(\sigma_1) = \alpha(D[1]) \\ [\alpha(\sigma_1)|\alpha(D)] & \text{otherwise} \end{cases} \end{aligned}$$

Let D be an ω_r derivation, then $\alpha(D)$ is a valid ω_t derivation (it is easy to verify that the last state in $\alpha(D)$ is indeed a final state). \square

Determinism While ω_r clearly is far more deterministic than ω_t , two sources of nondeterminism can still be distinguished:

1. In the **Solve** transition, the concrete set of reactivated constraints is not fully determined (see earlier). Moreover, the order in which constraints are reactivated is left open.

³These bounds are not part of the original specification by Duck et al. (2004). They were introduced by Duck (2005) and Schrijvers (2005) to better reflect actual behaviour of CHR implementations.

2. If multiple instances of the same rule (and matching the same active constraint) are applicable, the order in which they are found in **Propagate** and **Simplify** is not determined.

The motivation is to leave more freedom for efficient constraint store data structures and other optimisations to CHR implementations.

4.3 Program Properties and Analysis

In this section we discuss important properties of CHR programs: termination (Section 4.3.1), confluence (Section 4.3.2), and complexity (Section 4.3.3), as well as (semi-)automatic analysis of these properties.

4.3.1 Termination

Termination analysis attempts to determine whether the evaluation of a given program will terminate. Due to the following properties of CHR, in particular the last one (Frühwirth 2000), traditional techniques from logic programming and term rewriting cannot readily be applied:

- CHR rules are multi-headed
- CHR constraints have a multiset semantics
- CHR propagation rules do not remove any constraints

In recent years though, considerable progress has been made. So much so that our survey in (Sneyers et al. 2010, Section 3) is already outdated. Pilozzi and De Schreye (2008) developed innovative termination conditions that can prove most practical CHR programs terminating, including those that contain propagation rules. Their approach is strictly more powerful than that of Frühwirth (2000) and Voets et al. (2007). Building on these results, Pilozzi (2009a) has developed a system called CHRisTA (CHR Termination Analyser), a constraint-based automated termination analyser for CHR(LP). This approach scales to large programs (Pilozzi and De Schreye 2009).

Pilozzi (2009b) also proposed a novel constraint-based approach to termination analysis, applicable to both logic programming (LP) and CHR. Their approach elegantly deals with problems such as bounded increase and integer arithmetic, and is able to prove termination of programs that only terminate for subsets of the considered queries. This work constitutes a first crucial step towards a first integrated termination analyser for CHR(LP) (Pilozzi 2009c).

4.3.2 Confluence

A CHR program is called ω -confluent if, for any (initial) state, all ω derivations from that state result in equivalent final states.

In early CHR research, Abdennadher et al. (1997, 1999) derived a decidable, sufficient and necessary test for proving ω_t -confluence of terminating programs, and proved that confluence implies correctness (consistency of the logical reading; see Section 4.2). Their test essentially entails identifying all minimal execution states that constitute sources of nondeterminism (called *critical states*), and showing that all possible derivations from such states reconverge to an equivalent state. Based on this, Bouissou (2004) implemented a confluence analyser in CHR.

The notions of ω_r - and ω_p -confluence— ω_p is the semantics underlying CHR^{FP}: see Section 4.4.2—were investigated respectively by Duck (2005, Chap. 6) and De Koninck (2008, Chap. 3). Both studies also discuss practical confluence tests.

Recently, the topic of ω_t -confluence received renewed attention. Raiser and Tacchella (2007) consider confluence of non-terminating programs, and Duck et al. (2007) introduce the powerful notion of *observable confluence*. Many ‘non-confluent’ programs are observably confluent, because apparent non-confluence originates from critical states that are unreachable in practice. To properly analyse observable confluence though, more powerful analysers are required capable of reasoning over various types of constraint store *invariants*.

Related properties and analyses

Monotonicity The completeness of confluence tests inherently relies on the *monotonicity* property of CHR programs. In its pure form, all CHR programs are *monotonic*: that is, all applicable rules in a given state remain applicable if either CHR or built-in constraints are added to the constraint store.

Completion Completion is a technique to transform a non-confluent CHR program into a confluent one (Abdennadher and Frühwirth 1998), useful for the extension, modification and specialisation of existing programs. Frühwirth (2005b) for instance showed related completion techniques can improve the parallelizability of CHR programs.

Operational equivalence Two programs are *operationally equivalent* if they reach equivalent results for each query (Abdennadher and Frühwirth 1999; Abdennadher 2001). Based on operational equivalence, Abdennadher and Frühwirth (2004) presented a method to remove redundant rules from CHR programs, as well as techniques to merge two CHR solvers.

4.3.3 Complexity

Computational complexity theory investigates the amount of computational resources needed to execute an algorithm; the two most important of which are time (execution time, number of steps) and space (size of the required memory).

Ad hoc analysis

The complexity of various specific, individual CHR programs has been analysed. Notable examples include the proven complexity-wise optimality of CHR implementations of several classical algorithms—such as union-find (Schrijvers and Frühwirth 2006) and Dijkstra’s algorithm (Sneyers et al. 2006a)—and the complexity analyses of several CHR-based constraint solvers by e.g. Frühwirth (2005a) and Meister et al. (2006) (cf. Section 4.6.1).

Meta-complexity results

While ad hoc methods give the most accurate results in practice, they cannot easily be generalised. Therefore, more structured approaches to complexity analysis have been proposed by means of meta-complexity theorems.

Building on the elementary CHR termination analysis techniques of (Frühwirth 2000), Frühwirth (2001, 2002a, 2002b) investigated the time complexity of simplification-only programs for naive implementations of CHR. Recent work on optimising compilation of CHR, however, allows meta-theorems that give much tighter complexity bounds. We now discuss two distinct approaches.

De Koninck et al. (2007a) establish a close correspondence between CHR and Ganzinger and McAllester (2002)’s Logical Algorithms (LA) formalism, allowing the LA meta-complexity result to be applied (indirectly) to a large class of CHR^{TP} programs (CHR^{TP} subsumes CHR: see Section 4.4.2).

Sneyers et al. (2009) introduce the *CHR machine*, a new model of computation similar to the well-known Turing and RAM machine, and establish strong meta-complexity results. Like Frühwirth (2001, 2002a, 2002b), Sneyers et al. (2009) first estimate the number of rule applications (CHR machine steps). In a second step, however, they additionally compute the complexity of individual rule applications. While the first step only depends on the operational semantics of CHR, the second strongly depends on the performance of the actual code generated by the CHR compiler. With this approach, Sneyers et al. (2009) obtain tight bounds for both time and space complexity.

Complexity-wise completeness of CHR

The most interesting result of Sneyers et al. (2009) is the following complexity-wise completeness result for CHR (details omitted):

“ For every algorithm [...], a CHR program exists which can be executed by [an optimising CHR system] with optimal time and space complexity. ”

Complexity-wise completeness implies Turing completeness but is a much stronger property. Sneyers et al. (2009) argue CHR is the first declarative language for which such a result can be demonstrated within the pure part of the language, namely without imperative extensions (i.e. under ω_t).

4.4 Language Extensions

Over the years, weaknesses and limitations of CHR have been identified, for instance regarding execution control, expressiveness, modularity, incrementality, and search. In this section we consider extensions and variants of CHR that were proposed to tackle these issues.

4.4.1 Probabilities

Probabilistic CHR or PCHR (Frühwirth, Di Pierro, and Wiklicky 2002) extends CHR with probabilistic choice between the applicable rules in any state.

Example 4.12. The following PCHR program simulates a fair coin toss:

```
throw(Coin) <=>0.5: Coin=head.
throw(Coin) <=>0.5: Coin=tail.
```

PCHR is implemented by means of a straightforward source-to-source transformation using the framework of Frühwirth and Holzbaaur (2003).

Sneyers et al. (2009) identify several important conceptual issues of PCHR, and propose the novel rule-based probabilistic-logic formalism CHRiSM (Chance Rules induce Statistical Models). CHRiSM is a particularly powerful, expressive formalism, that allows elegant embeddings of many probabilistic logic formalisms. CHRiSM has a cleaner, more natural semantics than PCHR, as the probabilistic meaning of CHRiSM rules is local, that is, it does not depend on the full program and runtime information. CHRiSM is implemented using a source-to-source transformation to CHR(PRISM). PRISM is an established probabilistic extension of Prolog. CHRiSM directly inherits PRISM's impressive support for probabilistic inference tasks, such as sampling, probability computation, and an expectation-maximisation (EM) learning algorithm.

4.4.2 Priorities

While the refined operational semantics reduces most of the nondeterminism of the ω_t semantics, it arguably does not offer the CHR programmer an intuitive and predictable way to influence control flow. The ω_r semantics in a sense forces the programmer to understand and take into account how CHR implementations work, to achieve the desired execution control.

CHR^{IP} gives the programmer a much more precise and high-level control over program execution (De Koninck et al. 2007b; De Koninck 2008). In CHR^{IP}, the programmer assigns a *priority* to every rule. A rule's priority is either a numeric constant (*static priority*) or an arithmetic expression involving variables appearing in the rule's head (*dynamic priority*). CHR^{IP}'s semantics ensures that

3. Apply. $\langle \emptyset, K \sqcup R \sqcup S, \mathbb{B}, \mathbb{T} \rangle_n \xrightarrow{\mathcal{P}} \langle B, K \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T} \sqcup \{t\} \rangle_n$
 where $p :: \rho @ H_k \setminus H_r \Leftrightarrow G \mid B$ is a renamed apart rule of \mathcal{P} ,
 $\theta \in \text{matchings}(H_k, H_r, G, K, R, \mathbb{B})$, and $t = (\rho, \text{ID}(K) ++ \text{ID}(R)) \notin \mathbb{T}$.
 Furthermore, no rule instance with instantiated priority expression $\theta'(p')$
 exists for which these conditions hold, and for which $\theta'(p') > \theta(p)$.

Figure 4.3: The **Apply** transition rule of the priority semantics ω_p

rule instances with higher priority are applied first. Dynamic priorities allow different instances of the same rule to be executed at different priorities.

Example 4.13. An example that illustrates the power of dynamic priorities is the following CHR^{FP} implementation of Dijkstra’s algorithm:

```

1 :: source(V) ==> dist(V,0).
1 :: dist(V,D1) \ dist(V,D2) <=> D1 ≤ D2 | true.
D+2 :: dist(V,D), edge(V,C,U) ==> dist(U,D+C).

```

The priority of the last rule makes sure that new distance labels are propagated in the right order: first the nodes closest to the source node.

Formally, CHR^{FP}’s semantics, denoted ω_p , is a variant of ω_t where the **Apply** transition rule is replaced with the version depicted in Figure 4.3. Besides from the obvious priority restriction, there is another important difference: the ω_p **Apply** rule is only applicable on states with an empty goal. In other words, all CHR constraints have to be **Introduced**, and all built-in constraints **Solved**, before the next CHR rule is allowed to fire. Without this so-called *batch semantics*, the execution control provided by rule priorities would become too nondeterministic. By first moving all constraints to their respective constraint stores, the ω_p avoids that low-priority rules are allowed to fire only because the constraints required in higher-priority matches are still in the goal.

De Koninck (2008) furthermore defined a refined operational semantics ω_{rp} for CHR^{FP}, developed an optimising compiler, and studied language properties such as ω_p -confluence and complexity (see also Section 4.3).

4.4.3 Adaptive CHR

Constraint solving in a continuously changing, dynamic environment often requires immediate adaptation of the solutions, i.e. when constraints are added or removed. By nature, CHR solvers already support efficient adaptation when constraints are added. Wolf (1999, 2000, 2000a) introduces an extended incremental adaptation algorithm which is capable of adapting CHR derivations after constraint retractions as well. An efficient implementation exists in Java (Wolf

2001a, 2001b; cf. Section 4.5.3). Interesting applications of adaptive CHR include adaptive solving of soft constraints, discussed in Section 4.6.1, and the realization of intelligent search strategies, discussed in Sections 4.4.4 and 4.5.3.

4.4.4 Disjunction and search

Next to constraint simplification and propagation, most constraint solvers require search. Pure CHR though does not offer any support for search. Abdennadher and Schütz (1998) therefore proposed an extension of CHR with disjunctions in rule bodies (see also Abdennadher 2000, 2001). The resulting language is denoted CHR^\vee (pronounced “CHR or”), and is capable of expressing several declarative evaluation strategies, including both bottom-up and top-down evaluation, model generation and abduction (abduction is discussed in Section 4.6.3). Any (pure) Prolog program can be rephrased as an equivalent CHR^\vee program (Abdennadher 2000, 2001). An interesting aspect of CHR^\vee is that the extension comes for free in CHR(Prolog) implementations by means of the built-in Prolog disjunction and search mechanism.

As a typical example of programming in CHR^\vee , consider the following rule:

```
labelling, X::Domain <=> member(X,Domain), labelling.
```

Note the implicit don't known choice in the call to Prolog's `member/2` predicate.

Various ways have been proposed to make the search in CHR^\vee programs more flexible and efficient. Menezes, Vitorino, and Aurelio (2005) present a CHR^\vee implementation for Java in which the search tree is made explicit and manipulated at runtime to improve efficiency. De Koninck et al. (2006b) extend both ω_t and ω_r towards CHR^\vee . The theoretical semantics leaves the search strategy undetermined, whereas the refined one allows the specification of various search strategies. They also realised an implementation for different strategies in CHR(Prolog) by means of a source-to-source transformation.

For CHR(Java) systems, of course, the host language does not provide built-in search capabilities. The specification of intelligent search strategies, i.e. more flexible and powerful than Prolog's built-in chronological backtracking, has therefore received considerable attention in several CHR(Java) systems (Krämer 2001; Wolf 2005). As described in Section 4.5.3, in these systems, the search strategies are implemented and specified in Java, orthogonally to the actual CHR program. Wolf, Robin, and Vitorino (2007) propose an implementation of CHR^\vee using the ideas of (Wolf 2005), in order to allow a more declarative formulation of search in the bodies of CHR rules, while preserving efficiency and flexibility. Along these lines is the approach proposed by Robin, Vitorino, and Wolf (2007), where disjunctions in CHR^\vee would be transformed into special purpose constraints that can be handled by an external search component such as JASE (Krämer 2001; cf. Section 4.5.3).

4.5 Systems and Implementation

CHR is first and foremost a programming language. Hence, a large part of CHR research has been devoted to the development of CHR systems and efficient execution of CHR programs. In this section, we provide a first overview of these contributions. A comprehensive discussion of CHR compilation and optimisation techniques is given in Part III of this dissertation.

Many CHR systems (compilers, interpreters and ports) have been developed, for many different host languages. Figure 4.4 presents a time line of system development, branches and influences. We now discuss these systems, grouped by host language or host paradigm, in more detail.

4.5.1 CHR(LP)

Logic Programming is the natural host language paradigm for CHR. Hence, it is not surprising that the CHR(Prolog) implementations are the most established.

The seminal work by Holzbaaur and Frühwirth (1999, 2000a) has laid the groundwork for an entire generation of CHR systems. They adapted the compilation scheme of an earlier CHR system for ECL²PS^e Prolog (Frühwirth and Brisset 1995) to use for the first time the efficient, flexible attributed variables feature found in SICStus Prolog. For a long time, the CHR(SICStus) system by Holzbaaur was considered the reference CHR implementation. Its efficient compilation scheme served as the basis for many other systems, and was formalised in the refined operational semantics (Duck et al. 2004; cf. Section 4.2.3).

The K.U.Leuven CHR system by Schrijvers and Demoen (2004b) started as a reconstruction of Holzbaaur's CHR(SICStus) system, created as a benchmark for Demoen (2002)'s dynamic attributed variables implementation in hProlog. It soon became clear there was much potential for improvement, and the system gradually diverged from its roots. It became the central topic of Schrijvers's seminal Ph.D. thesis on analyses, optimizations and language extensions of CHR. Schrijvers (2005) made numerous contributions to the field of optimising compilation of CHR (cf. Chapter 8). K.U.Leuven CHR is currently available in XSB (Schrijvers et al. 2003; Schrijvers and Warren 2004), SWI-Prolog (Schrijvers, Wielemaker, and Demoen 2005), YAP, B-Prolog (using Action Rules; Schrijvers, Zhou, and Demoen 2006), SICStus 4 (replacing Holzbaaur's system) and Ciao Prolog. Another system directly based on the work of Holzbaaur and Schrijvers is the CHR library for the linear logic concurrent constraint programming language SiLCC by Bouissou (2004). All of these systems compile CHR to host language code. The only *interpreter* for CHR(Prolog) is TOYCHR⁴.

HAL is a constraint logic programming language designed to support the construction and extension and use of new constraint solvers. The CHR(HAL)

⁴by Gregory J. Duck, 2003. Download: <http://www.cs.mu.oz.au/~gjd/toychr/>

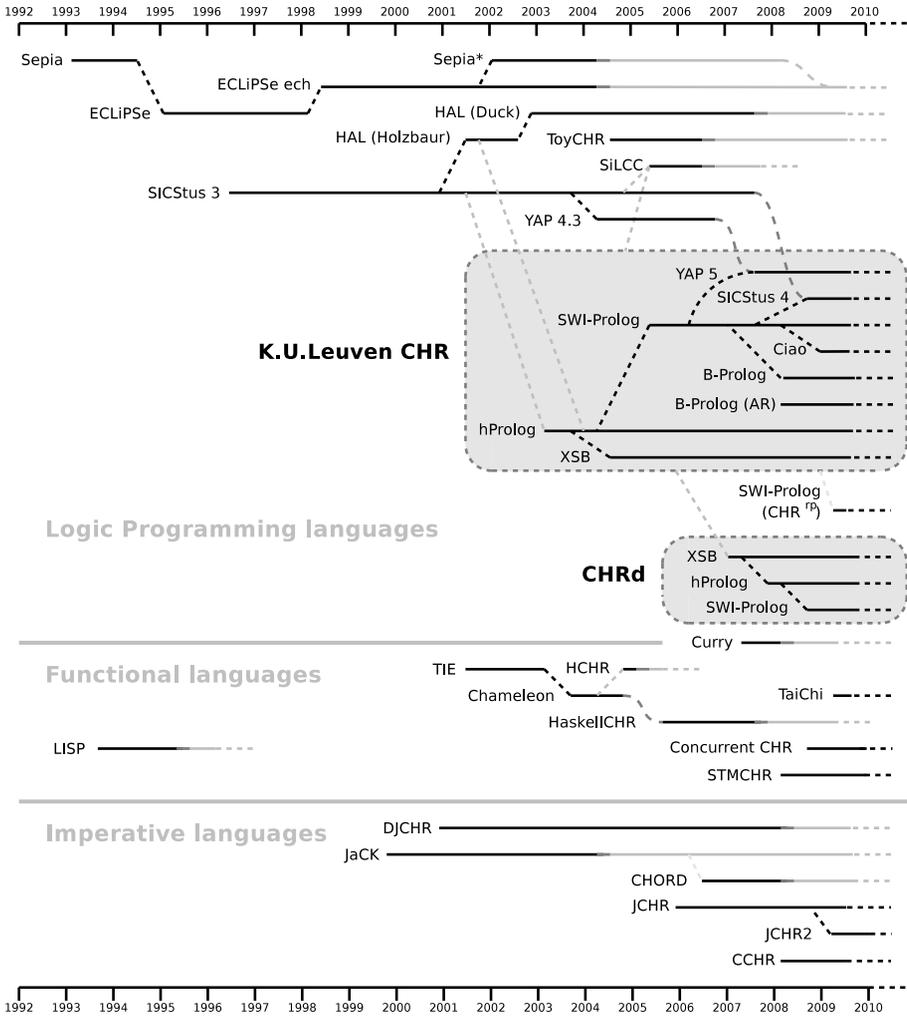


Figure 4.4: A timeline of CHR implementations.

system by Holzbaur, García de la Banda, Stuckey, and Duck (2005), was developed in parallel with, and based on much the same principles as, the K.U.Leuven CHR system. Many of the now standard CHR analyses and optimizations (see Chapter 8) were developed collaboratively by Duck (2005) and Schrijvers (2005). HALCHR also pioneered the interaction of CHR with arbitrary built-in constraint solvers (Duck et al. 2003).

More recently, systems with deviating operational semantics have been developed. The CHRd system by Sarna-Starosta and Ramakrishnan (2007) runs in XSB, SWI-Prolog and hProlog. It features a constraint store with set semantics and is particularly suitable for tabled execution. The CHR^{FP} system by De Koninck (2008) for SWI-Prolog provides rule priorities. We further build on the foundational work by De Koninck (2008), discussed in Section 4.4.2, throughout this dissertation.

4.5.2 CHR(FP)

As type checking is one of the most successful applications of CHR in the context of Functional Programming (see Section 4.6.3), several CHR implementations were developed specifically for this purpose. Most notable is the Chameleon system (Stuckey and Sulzmann 2005) which features CHR as the programming language for its extensible type system. Internally, Chameleon uses the HaskellCHR implementation⁵. The earlier HCHR prototype (Chin, Sulzmann, and Wang 2003) had a rather heavy-weight and impractical approach to logical variables (cf. Section 7.4.3).

The aim of a 2007 Google Summer of Code project was to transfer this CHR-based type checking approach to two Haskell compilers (YHC and nhc98). The project led to a new CHR(Haskell) interpreter called TaiChi (Boespflug 2007).

With the advent of software transactional memories (STM) in Haskell, two prototype systems with parallel execution strategies have been developed: STM-CHR (Stahl and Melnikov 2007) and Concurrent CHR (Sulzmann and Lam 2008). Sulzmann and Lam (2007a) also explored the use of Haskell's laziness and concurrency abstractions for implementing the search of partner constraints. These systems are currently the only known CHR implementations that exploit the inherent parallelism in CHR programs. Concurrent CHR also serves as the basis for Haskell-Join-Rules (Sulzmann and Lam 2007b; cf. Section 4.1.4).

We also mention the Haskell library for the PAKCS implementation of the functional logic language Curry (Hanus 2006). The PAKCS system actually compiles Curry code to SICStus Prolog, and its CHR library is essentially a front-end for the SICStus Prolog CHR library. The notable added value of the Curry front-end is the (semi-)typing of the CHR code.

⁵by Gregory J. Duck, 2004. Download: <http://www.cs.mu.oz.au/~gjd/haskellchr/>

4.5.3 CHR(Java) and CHR(C)

CHR systems are available for the imperative host languages Java and C. These multi-paradigmatic integrations of CHR and mainstream programming languages offer powerful synergetic advantages to the software developer: they facilitate the development of application-tailored constraint systems that cooperate efficiently with existing components. These advantages, as well as the different conceptual and technical challenges encountered when embedding CHR into an imperative host language, are discussed further in Chapter 7.

There are at least four implementations of CHR in Java. The earliest is the Java Constraint Kit (JaCK) by Abdennadher (2001) and others (Abdennadher, Krämer, Saft, and Schmauß 2002). The JaCK framework consists of three major components:

1. JCHR (Schmauß 1999), which uses a CHR dialect intended to resemble Java, in order to provide an intuitive programming experience. No operational semantics is specified for this system, and its behaviour deviates from other CHR implementations.
2. VisualCHR (Abdennadher and Saft 2001), an interactive tool visualising the execution of JCHR, briefly discussed in Section 4.5.4.
3. JASE (Krämer 2001), the Java Abstract Search Engine, which allows the specification of tree-based search strategies. The JASE library is added to the JaCK framework as an orthogonal component. It provides a number of utility classes that aid the user to implement search algorithms in the Java host language. A typical algorithm consists of the following two operations, executed in a loop: a JCHR handler is run until it reaches a fix-point, after which a new choice is made. If an inconsistency is found, backtracking is used to return to the previous choice point. JASE aids in maintaining the search tree, and can be configured to use either trailing or copying to facilitate backtracking.

The original CHORD system (Constraint Handling Object-oriented Rules with Disjunctive bodies)⁶, developed as part of the ORCAS project (Robin and Vitorino 2006), was a Java implementation of CHR^v (Menezes, Vitorino, and Aurelio 2005). Its implementation was built on that of JaCK, adding the possibility to use disjunctions in rule bodies.

Both JaCK (Schmauß 1999) and CHORD take a different approach compared to most other CHR compilers. After some preprocessing by the front-end, CHR programs are essentially interpreted. No major optimisations are performed. The main issue with these systems are their lacking performance. In (Van Weert et al. 2005), for instance, JaCK was shown many orders of magnitude slower than state-of-the-art systems.

⁶by Jairson Vitorino and Marcos Aurelio, 2005, no longer available

DJCHR (Dynamic JCHR; Wolf 2001a) is an implementation of adaptive CHR (see Section 4.4.3). DJCHR uses a compilation scheme similar to the basic CHR(Prolog) scheme, where the underlying incremental adaptation algorithm maintains *justifications* for rule applications and constraint additions. Building on the approach of Holzbaur and Frühwirth (1999, 2000a), fast partner constraint retrieval is achieved using a form of attributed variables (Wolf 2001b). Wolf (2005) shows that DJCHR’s justifications, and in particular those of any derived **false** constraint, also constitute an effective basis for intelligent search strategies. As in JaCK, the different search algorithms are implemented orthogonally to the CHR program. Wolf’s approach confirms that advanced search strategies are often more efficient than a low-level, built-in implementation of chronological backtracking (as in Prolog).

The K.U.Leuven JCHR systems focus both on performance and on integration with the host language, two areas where earlier CHR(Java) systems were particularly lacking. K.U.Leuven JCHR handlers integrate neatly with existing Java code, and it is currently one of the most efficient CHR systems available. The language design and compilation techniques of the JCHR and JCHR2 systems are the main topic of this dissertation.

CCHR (Wuille et al. 2007) implements CHR for C. It is heavily inspired by JCHR. It is an extremely efficient CHR system conforming to the ω_r refined operational semantics. We discuss CCHR in more detail in Section 7.4.1.

4.5.4 Programming Environments

Over the past decade there has been an exponential increase in the number of CHR systems, and CHR compilation techniques have matured considerably. The support for advanced software development tools, such as debuggers, refactoring tools, and automated analysis tools, lags somewhat behind, and remains an important challenge for the CHR community (cf. Section 11.2).

VisualCHR (Abdennadher and Saft 2001), part of JaCK (see Section 4.5.3), is an interactive tool visualising the execution of CHR rules. It can be used to debug and to improve the efficiency of constraint solvers.

Both Holzbaur’s CHR implementation and the K.U.Leuven CHR system feature a trace-based debugger that is integrated in the Prolog four port tracer. A generic trace analysis tool, with an instantiation for CHR, is presented by Ducassé (1999). As briefly discussed in Section 7.3.6, the K.U.Leuven JCHR systems also provide rudimentary trace-based debuggers.

Schumann (2002) presents a literate programming system for CHR. The system allows for generating from the same literate program source both an algorithm specification typeset in \LaTeX using mathematical notation, and the corresponding executable CHR source code.

4.6 Applications

Early successful applications of CHR include the optimal placement of wireless transmitters (Frühwirth and Brisset 1998, 2000), and the *Munich Rent Advisor*, an expert system for estimating the maximum fair rent in the city of Munich (Frühwirth and Abdennadher 2001). In this section, we give an overview of the many and diverse recent applications of CHR. A more thorough, exhaustive survey can be found in (Sneyers, Van Weert, Schrijvers, and De Koninck 2010).

4.6.1 Constraint solvers

CHR was originally designed specifically for writing constraint solvers. Recent examples of non-trivial constraint solvers written in CHR include:

Soft constraints Bistarelli et al. (2004) present a series of constraint solvers for finite domain *soft constraints* based on the framework of *c-semirings*. A node and arc consistency solver is presented, as well as complete solvers based on variable elimination or branch and bound optimisation.

Constraint hierarchies Wolf (2000b) proposes an approach for solving dynamically changing over-constrained problems, modelled using (finite domain) constraints with hierarchical strengths or preferences using an adaptive CHR solver (Wolf et al. 2000; Wolf 2001a; see Section 4.4.3).

Interactive constraint satisfaction Alberti et al. (2005) describe the implementation of a CLP language for expressing Interactive Constraint Satisfaction Problems (ICSP). In the ICSP model incremental constraint propagation is possible even when variable domains are not fully known.

Lexicographic order Frühwirth (2006a) presented a constraint solver for a general lexicographic order constraint in terms of inequality constraints offered by the underlying solver.

Trees Meister et al. (2006) developed a solver for conjunctions of non-flat equations over rational trees. Djelloul et al. (2007) use it in a more general solver for (quantified) first-order constraints over finite and infinite trees.

Sequences Kosmatov (2006a, 2006b) has constructed a constraint solver for sequences, expressing many sequence constraints in terms of two basic constraints, for sequence concatenation and size.

Non-linear constraints A general purpose CHR-based CLP system for non-linear (polynomial) constraints over the real numbers was presented by De Koninck et al. (2006a). The system, called INCLP(\mathbb{R}), is based on interval arithmetic and uses an interval Newton method and constraint inversion to achieve respectively box and hull consistency. INCLP(\mathbb{R}) is part of the standard SWI Prolog distribution (Wielemaker et al. 2010).

CHR is ideally suited to create application-specific constraint solvers, as witnessed by the applications listed in the upcoming subsections.

Scheduling

Abdennadher and Marte (2000) successfully used CHR for scheduling courses at the university of Munich. A related problem, namely that of assigning classrooms to courses, is dealt with by Abdennadher, Saft, and Will (2000). An overview of both applications is found in (Abdennadher 2001).

Spatio-temporal reasoning

In autonomous mobile robot navigation, a crucial topic is automated qualitative reasoning about spatio-temporal information, including orientation, distances, directions, topology and time. The use of CHR for spatio-temporal reasoning has received extensive research attention. Particularly noteworthy are the contributions of Escrig and Toledo (1998a, 1998b) and Cabedo and Escrig (2003).

Meyer (2000) has applied CHR for the constraint-based specification and implementation of diagrammatic environments. CHR allows the integration with other constraint domains, and extra CHR rules can be added to model more complex diagrammatic systems. Similar results are obtained with CHR_G in the context of natural language processing (see Section 4.6.3).

Multi-agent systems

FLUX (Thielscher 2002, 2005) is a high-level programming system, implemented in CHR and based on fluent calculus, for cognitive agents that reason logically about actions in the context of incomplete information. An interesting application of this system is FLUXPLAYER (Schiffel and Thielscher 2007), which won the 2006 General Game Playing (GGP) competition at AAAI'06. Seitz, Bauer, and Berger (2002) and Alberti et al. (2003, 2004, 2006) also successfully applied CHR-based reasoning in the context of multi-agent systems.

Lam and Sulzmann (2006) explore the use of CHR as an agent specification language, founded on CHR's linear logic semantics (see Section 4.2.1). They introduce a monadic operational semantics for CHR, where special *action constraints* have to be processed in sequence (discussed also in Section 5.1.5).

Semantic web and Web 3.0

A core problem related to the *Semantic Web* is the integration and combination of data from diverse heterogeneous information sources. CHR has proven effective in the implementation of several powerful mediator-based integration systems (Bressan and Goh 1998; Firat 2003; Badea, Tilivea, and Hotaran 2004).

The Cuyper Multimedia Transformation Engine (Geurts, van Ossenbruggen, and Hardman 2001) is a prototype system for automatic generation of Web-based presentations adapted to device-specific capabilities and user preferences. It uses CHR and traditional CLP to solve spatio-temporal constraints.

Automatic Generation of Solvers

Many authors have investigated the automatic generation of CHR-based constraint solvers from formal specifications.

A first line of work is that of Apt and Monfroy (2001). From an extensional definition of a finite domain constraint, a set of propagation rules is derived. As an extension, Brand and Monfroy (2003) propose to transform the derived rules to stronger propagation rules, for specialised versions of constraints.

A second is that of Abdennadher and Rigotti (2004), who derive both propagation and specialised simplification rules. Abdennadher et al. (2006) implemented this algorithm in the Automatic Rule Miner tool. In later work, Abdennadher et al. (2005, 2008) further improved this approach, and extended it to intentional definitions where constraints are defined by logic programs.

Brand (2002) proposed a method to eliminate redundant propagation rules and applies it to rules generated by the algorithm of Abdennadher et al. (2006).

4.6.2 Algorithms

CHR is increasingly used as a general-purpose programming language. Schrijvers and Frühwirth (2005a, 2006) implemented and analysed the classic union-find algorithm in CHR. In particular, they showed how the optimal complexity of this algorithm can be achieved in CHR—a non-trivial achievement since this is believed to be impossible in pure Prolog (cf. Section 4.3.3). This work led to parallel versions of the union-find algorithm (Frühwirth 2005b) and several derived algorithms (Frühwirth 2006b). This work inspired Sneyers et al. (2009) to study the complexity of CHR-based algorithms in general, leading to their fundamental complexity-wise completeness result (cf. Section 4.3.3).

Other examples of elegant and natural CHR implementations of classic algorithms include Dijkstra's algorithm using Fibonacci heaps (Sneyers et al. 2006a), the preflow-push maximal flow algorithm (Meister 2006), and Hopcroft's algorithm for minimising states in a finite automaton (Sneyers 2008).

4.6.3 Programming language development

Type systems

CHR's aptness for symbolic constraint solving has led to many applications in the context of type system design, type checking and type inference. While the basic

Hindley-Milner type system requires only a simple Herbrand equality constraint, more advanced type systems require custom constraint solvers.

The most successful use of CHR in this area is for Haskell type classes. Type classes are a principled approach to ad hoc function overloading based on type-level constraints. By defining these type class constraints in terms of a CHR program (Stuckey and Sulzmann 2005), the essential properties of the type checker (soundness, completeness and termination) can easily be established. Various extensions, such as multi-parameter type classes (Sulzmann et al. 2006) and functional dependencies (Sulzmann et al. 2007) are easily expressed. At several occasions Sulzmann argues for HM(CHR), where the programmer can directly implement custom type system extensions in CHR.

Coquery and Fages (2003, 2005) presented a CHR-based type checker for Prolog and CHR(Prolog) that deals with parametric polymorphism, subtyping and overloading. Schrijvers and Bruynooghe (2006) reconstruct type definitions for untyped functional and logic programs.

Finally, Chin et al. (2006) presented a control-flow-based approach for variant parametric polymorphism in Java.

Abduction

Abduction is the inference of a cause to explain a consequence: given B determine A such that $A \rightarrow B$. It has applications in many areas: diagnosis, recognition, natural language processing, type inference, etc.

The HYPROLOG system of Christiansen and Dahl (2005a) integrates the early approach of Abdennadher and Christiansen (2000) with abductive-based logic programming. Both the abducibles and the assumptions are implemented as CHR constraints. Gavanelli et al. (2003) introduce two complementary approaches to implementing abductive logic programming using CHR. The system of Alberti et al. (2005) extends the abductive reasoning procedure with the dynamic acquisition of new facts.

Computational linguistics

CHR allows flexible combinations of top-down and bottom-up computation (Abdennadher and Schütz 1998), and abduction fits naturally in CHR as well (see Section 4.6.3). It is therefore not surprising that CHR has proven a powerful implementation and specification tool for language processors.

Penn (2000) focuses on another benefit of CHR, namely the possibility of delaying constraints until their arguments are sufficiently instantiated. As a comprehensive case study he considers a grammar development system for HPSG, a popular constraint-based linguistic theory.

Morawietz and Blache (2002) show that CHR allows a flexible and perspicuous implementation of a series of standard chart parsing algorithms (cf. also Morawietz

(2000)), as well as more advanced grammar formalisms. Along the same lines is the CHR implementation of a context-sensitive, rule-based grammar formalism by Garat and Wonsever (2002).

A recent application of CHR in the context of natural language processing is (Christiansen and Have 2007), where a combination of Definite Clause Grammars (DCG) and CHR is used to automatically derive UML class diagrams from use cases written in a restricted natural language.

CHR Grammars The most successful approach to CHR-based language processing is given by CHR grammars (CHRG), a highly expressive, bottom-up grammar specification language proposed by Christiansen (2005). Contrary to other approaches, which mostly use CHR as a general-purpose implementation language, Christiansen recognises that the CHR language itself can be used as a powerful grammar formalism. CHRG's, built as a relatively transparent layer of syntactic sugar over CHR, are to CHR what DCG's are to Prolog.

CHRG's inherent support for context-sensitive rules, combined with extragrammatical hypotheses modelled as regular CHR constraints, readily allow the natural modelling of advanced linguistic phenomena and grammar formalisms (Christiansen 2005; Aguilar-Solis and Dahl 2004; Dahl 2004).

Using CHRG, Dahl and Blache (2005) develop directly executable specifications of property grammars. Applications of this approach and its extensions Dahl and Voll (2004) include the extraction of concepts and relations from biomedical texts (Dahl and Gu 2006), early lung cancer diagnosis (Barranco-Mendoza 2005, Chapter 4), error detection and correction of radiology reports obtained from speech recognition (Voll 2006, Section 5.2.8), and the analysis of biological sequences (Bavarian and Dahl 2006).

Christiansen and Dahl (2003) use an abductive model based on CHRG to diagnose and correct grammatical errors. Other applications of CHRG include the characterisation of the grammar of ancient Egyptian hieroglyphs (Hecksher, Nielsen, and Pigeon 2002), linguistic discourse analysis (Christiansen and Dahl 2005b), and the disambiguation of biological text (Dahl and Gu 2007). An approach similar to CHRG is taken by Bès and Dahl (2003) for the parsing of balanced parentheses in natural language.

Testing and verification

Another application domain for which CHR has proved useful is software testing and verification. Ribeiro, Zúquete, Ferreira, and Guedes (2000) present a CHR-based tool for detecting security policy inconsistencies. Lötzbeyer and Pretschner (2000) and Pretschner et al. (2004) propose a model-based testing methodology, in which test cases are automatically generated from abstract models using CLP and CHR. They consider the ability to formulate arbitrary test case specifications

by means of CHR to be one of the strengths of their approach. Gouraud and Gotlieb (2006) use a similar approach for the automatic generation of test cases for the Java Card Virtual Machine (JCVM).

4.6.4 Industrial CHR use

Although most CHR systems are still research prototypes, there are a few systems that can be considered to be robust enough for industrial application. We give a few examples of companies that are currently using CHR.

The New-Zealand-based company Scientific Software & Systems Ltd. is one of the main industrial users of CHR. CHR is used throughout its flagship product the Securitease stock broking system⁷. Securitease provides front office (order entry) and back-office (settlement and delivery) functions for stock brokers in Australia and New Zealand. Inside Securitease CHR is used for:

1. implementing the logic to recognise advantageous market conditions to automatically place orders in equity markets,
2. translating high-level queries to SQL,
3. describing complex relationships between mutually dependent fields on user input screens, and calculating the consequences of user input actions, and
4. realising a Financial Information eXchange (FIX) server.

The Canadian company Cornerstone Technology Inc.⁸ has created an inference engine for solving and optimising collections of design constraints, using Prolog and CHR. The design constraints work together to determine what design configuration to use, select components from catalogues, compute dimensions for custom components, and arrange the components into assemblies. The engine allows for generating, interactive editing, and validating of injection mould designs. Part of the system is covered by US Patent 7,117,055.

BSSE System and Software Engineering⁹, a German company specialising in the discipline of full automation of software development, uses CHR for the generation of test data for unit tests.

Agitar Technologies¹⁰ use the K.U.Leuven JCHR System in their flagship product family AgitarOne, a comprehensive JUnit testing product for Java. This application is discussed in some detail in Section 7.3.7.

At the MITRE Corporation¹¹, CHR is used in the context of optical network design. It is used to implement constraint-based optimisation, network configuration analysis, and as a tool coordination framework.

⁷<http://www.securitease.com/>

⁸<http://www.cornerstonemould.com/>

⁹<http://www.bsse.biz/>

¹⁰<http://www.agitar.com/>

¹¹<http://www.mitre.org/>

Part II

CHR Language Design

Chapter 5

A Next Generation CHR Language

When I first came here, this was all swamp. Everyone said I was daft to build a castle on a swamp, but I built it all the same, just to show them. It sank into the swamp. So I built a second one. That sank into the swamp. So I built a third. That burned down, fell over, then sank into the swamp. But the fourth one stayed up. And that's what you're going to get, Lad, the strongest castle in all of England.

— Michael Palin as **King of Swamp Castle**
in *Monty Python and the Holy Grail* (1975)

CHR aims at supporting a very high-level, declarative programming style. Adhering to the classic “what, not how” maxim of declarative programming considerably shortens development time, and vastly improves a program’s understandability, maintainability and robustness. Practical experience with current CHR systems, however, has made it increasingly clear that their syntax, language features, and semantics is insufficiently practical and declarative. CHR’s syntax has hardly evolved since its initial conception in 1992. The syntax though is somewhat archaic and overly verbose, and common tasks require cumbersome low-level encodings. Moreover, the currently prevalent operational semantics ω_r (cf. Section 4.2.3) is almost fully deterministic, much like an imperative language. De Koninck (2008) also incontestably established that more declarative means of exerting execution control are required.

In this chapter, we describe—for lack of a better name—CHR2, a solid basis for a next generation of CHR systems. The goals for CHR2 are as follows:

1. maintain the desirable features that made existing systems successful;
2. integrate the advantages of recent advances towards more declarative and useful CHR systems;

3. avoid any disadvantages of these earlier approaches that have since become apparent.

In Section 5.1 we introduce and motivate the principle features of CHR2. We focus on the design of a more modern, streamlined syntax, and the declarative specification of rules, execution control, and program invariants. We then formally specify the operational semantics of the core CHR2 language in Section 5.1, and study and discuss its properties in considerable detail.

Later, in Chapter 6, we seamlessly integrate negation and aggregates into CHR2, thus creating a particularly powerful, elegant programming language. The design of JCHR2, a reference implementation of CHR2 for Java, is discussed in Chapter 7, and its efficient compilation and optimisation in Part III.

5.1 Basic Building Blocks

5.1.1 Rule conditions

In conventional CHR syntax, the different types of *conditions* that determine a rule’s applicability—kept occurrences, removed occurrences, and guard conjuncts—are grouped in separate segments. Consequently, conditions that logically belong together must often be written separately. For larger, multi-headed rules, this hampers both usability and readability, as these restrictions mostly prohibit them from having an intuitive left-to-right reading.

Example 5.1. The following rule occurs in the RAM simulator program, used by Sneyers et al. (2009) to show the complexity-wise completeness of CHR (we reviewed this seminal work earlier in Section 4.3.3):

$$\text{prog}(L, \text{cJUMP}, R, _), \text{mem}(R, X) \setminus \text{pc}(L) \Leftrightarrow X \neq 0 \mid \text{pc}(L+1).$$

This rule simulates the CJUMP instruction of a *Random Access Machine*. The RAM machine’s program and memory are represented as `prog` resp. `mem` constraints; its program counter `L` is maintained in a single `pc` constraint.

The `mem/2` occurrence is written to the right of the `prog/4` occurrence because the latter logically determines the memory cell’s address `R`. The `pc/1` occurrence, however, similarly determines the instruction label `L` required for finding matching `prog/4` constraints. But because the `pc/1` occurrence is removed, it must unfortunately be written to the right of the backslash. The guard on `X` is similarly separated from the variable’s occurrence in the head.

This situation only further deteriorates when adding new types of rule conditions such as aggregates in Chapter 6.

We therefore propose a more flexible syntax. All rule conditions, including the conjuncts of the guard, are written on the left-hand side of the rule. Given

that in most systems all CHR constraints are explicitly declared—typically with (optional) mode and type information—a CHR compiler can normally easily distinguish between CHR and built-in constraints. Removed occurrences are preceded by a ‘-’ modifier. To clarify or disambiguate when needed, the ‘?’ modifier may be used for guard conjuncts, and ‘+’ for kept occurrences.

The arrow symbol ‘=>’ is used to separate a rule’s left- and right-hand side. The left-hand side now represents all the rule’s applicability conditions, the right-hand side contains the actions performed when a rule is applied.

Example 5.2. The rule from Example 5.1 can now be written as follows:

```
-pc(L), +prog(L,cJUMP,R,_), +mem(R,X), X ≠ 0 => pc(L+1).
```

The ‘+’ modifiers are optional, but we mostly include them as a matter of good programming style.

Rules without left-hand side are supported. Such rules are especially useful for specifying the constraints that constitute (part of) the initial constraint store. They constitute the counterparts of facts in Prolog, and can be thought of as representing logical implications with antecedent `true`. Similarly, for rules with trivial body `true`, the ‘=> `true`’ may be omitted¹.

Example 5.3. The following two rules illustrate idiomatic use of this syntax:

```
=> min(0).
+min(X), -min(Y), X ≤ Y.
```

Conclusion The new CHR2 syntax facilitates cleaner, more readable rules, as well as a smoother, more natural extension with new types of rule conditions (as shown e.g. in Chapter 6). It can moreover readily be used alongside traditional CHR syntax, where ‘=>’ implies a default ‘+’ for all occurrences, and similar defaults apply for simplification and simpagation rules.

5.1.2 Constraint identifiers

In most operational semantics and implementations of CHR, each constraint is assigned a unique identifier. We propose the programmer can access these identifiers explicitly, be it as an abstract data type. Identifiers can be used for matching, and can be compared using standard constraints such as =, <, ≥, etc. Other possible operations include `remove(id)`, which removes a constraint from the constraint store², and `alive(id)`, to test whether the constraint has been

¹Unfortunately, this is not possible in CHR(Prolog) systems, as the resulting conjunctions (cf. Example 5.3) would be interpreted by the Prolog parser as illegal redefinitions of ‘,’/2.

²Care must be taken when allowing this operation outside rule bodies, as it may break existing optimisations and analyses.

removed or not. We introduce identifiers mainly for use by expert users, and to facilitate source-to-source transformations for language extensions. Example transformations that would have benefited greatly from explicit identifiers include (Van Weert et al. 2006b) and (De Koninck et al. 2007b).

5.1.3 Constraint arguments

In certain cases, production rule (PR) syntax is much more practical and concise than that of CHR, particularly for CHR constraints with a larger number of arguments. We therefore incorporate similar syntactic sugar into CHR2. We briefly illustrate this by example:

Example 5.4. The MANNERS benchmark program is a PR program implementing a classic constraint optimisation problem, where the goal is to find an optimal, acceptable seating arrangement for guests at a dinner party.³

When translated into CHR, it contains several head conjuncts of the form: ‘seating(_, _, _, _, _, L_seat, _)’. One of the rules moreover looks like:

```
..., -seating(A,B,C,D,E,F,no) => seating(A,B,C,D,E,F,yes), ...
```

Clearly, CHR’s syntax scales poorly to larger arities. Both for pattern matching and modifications, programmers must include and/or count irrelevant arguments. Perhaps even worse, if arguments are added, removed or otherwise rearranged, changes must be made throughout the entire program.

A simple, elegant solution is obtained by extending the already required constraint declarations with the possibility of assigning names to constraint arguments (see for instance Example 5.16 later for a possible syntax). This then allows syntactic sugar e.g. of form ‘seating{seat2 = L_seat}’, and:

```
..., -seating{path_done = no} # S
      => modify(S, {path_done = yes}), ...
```

5.1.4 Priority constraints

As seen in Section 4.4.2, in CHR^{IP}, the priority assigned to each rule is an expression that evaluates to an integer number. For dynamic priorities, this expression may contain arguments of the constraints matched by the rule.

There are considerable downsides to this approach. Firstly, these priority numbers impose a total preorder over applicable rule instances, whereas the programmer mostly wants to enforce only a partial preorder. Unavoidably this frequently results in unintentional constraints on the execution order. Secondly, determining a suited priority number for a rule requires global knowledge of

³MANNERS is part of the “Texas benchmark suite” by Miranker et al. (1991). Another famous program in this collection is the WALTZ program we used earlier in Example 2.1.

numbers already assigned to other rules. Thirdly, adding new priority numbers may require renumbering. This leads to a programming style where gaps are left between successive priorities—i.e., the sequence (10, 20, 30, ...) is used rather than (1, 2, 3, ...). Note that the use of line numbers in early, ‘unstructured’ languages such as BASIC and FORTRAN led to similar issues. Especially for larger programs, CHR^{FP}’s priority numbers thus rapidly become problematic. Many production rule programs exhibit these issues as well (Miranker et al. 1991; IllationTM 2007).

Therefore, we propose that rules are no longer assigned numbers. Instead, each rule is assigned a symbolic *rule descriptor*, an arbitrary term that may contain variables occurring in the remainder of the head. This is a generalisation of the atomic rule names traditionally used in CHR systems. Next, *priority constraints* are specified over these descriptors. Supported constraints are =, <, ≤, > and ≥, where ‘larger’ means ‘higher priority’, i.e. ‘must fire before’.⁴ The operands of priority constraints are *rule patterns*, or sets thereof, that are matched with rule descriptors either statically or dynamically. A program’s priority constraints thus imply a *partial* preorder on rule instances.

We briefly introduce priority constraint declarations by example.

Example 5.5. For the classic LEQ program (cf. Listing 4.1), the following priority declaration could be used:

```
priority transitivity < {reflexivity, idempotence, antisymmetry}.
```

It declares that the `transitivity` rule has lower priority than the other rules. For LEQ, this is required for optimal performance and termination behaviour.

In general, rule descriptors can be arbitrary compound terms, even containing variables shared with the head (see further). The operands of priority constraints are (sets of) rule descriptors as well, and matching is used to determine the set of rules they apply to. The set of rule descriptors matching the first operand is always implicitly subtracted from the set matching the second operand. This facilitates the use of the familiar matching wildcard ‘_’. A more convenient shorthand for the above declarations is thus ‘`priority transitivity < _`’. Obviously, specifying multiple ‘... < _’ or ‘... > _’ constraints in a single CHR2 program would result in inconsistencies. Two special priorities, `lowest` and `highest`, can be used instead; for instance: ‘`priority transitivity = lowest`’. Obviously, this reintroduces non-locality issues. For larger programs, related rules can be grouped, and more local ‘`lowest`’ and ‘`highest`’ priorities can be introduced.

Example 5.6. Example 4.13 on page 48 showed De Koninck (2008)’s succinct CHR^{FP} implementation of Dijkstra’s shortest path algorithm:

⁴This is in contrast with CHR^{FP}, where a *higher* number indicates a *lower* priority (cf. Example 5.7). We find our interpretation more natural, but this is a matter of personal taste.

```

1 :: source(V) ==> dist(V,0).
1 :: dist(V,D1) \ dist(V,D2) <=> D1 ≤ D2 | true.
D+2 :: dist(V,D), edge(V,C,U) ==> dist(U,D+C).

```

The ‘D+2’ priority is somewhat artificial. And in larger programs, this approach would result in overlapping priority ranges. In CHR₂, this program becomes:

```

init          @ +source(V) => dist(V,0).
keep_shortest @ +dist(V,D1), -dist(V,D2), D1 ≤ D2.
label(D)      @ +dist(V,D), +edge(V,C,U) => dist(U,D+C).

priority keep_shortest > label(_),
          label(X) > label(Y) if X < Y.

```

From CHR₂’s priority constraints the intended priorities are readily apparent. Also, the constraints can be stated locally, independently of any other rules. This example further illustrates the declaration of dynamic priorities by including head arguments in the rule descriptors, and the ‘if’ construct to declare *conditional priority constraints*. Added advantage is that if, for instance, the `init` rule would require a priority *lower* than all `label(_)` instances, this could be declared in CHR₂ as: ‘`priority label(_) > init`’. Expressing this in CHR^{FP} is impossible, as no upper bound on the distance D is known a priori.

Example 5.7. For backwards compatibility, or for smaller programs, integer numbers can still be used to specify priorities. In CHR(Prolog) for instance, it then suffices to add the following declaration to regain CHR^{FP}’s semantics:

```

priority X > Y if ground(X), ground(Y), X < Y.

```

Related work Priorities are used by many related formalisms, such as production rule systems, term and graph rewriting, and constraint (logic) programming. These approaches either use integer numbers as in CHR^{FP}, or even only a fixed number of priority levels. An overview is given e.g. by De Koninck (2008, §3.9). The approach to rule preference closest to our symbolic constraint-based one seems to be that of preferred answer set programming (Brewka and Eiter 1999).

Conclusion Our symbolic probability constraints offer a more high-level, expressive and flexible execution control mechanism than CHR^{FP}’s numeric priority expressions. They achieve a better separation of logic and control, and allow partial orderings on rule priorities to be specified locally. In Section 6.3.2 we moreover argue they are better suited for automatic code generation, where all required priorities are not always known in advance.

5.1.5 Batch and sequential conjunctions

In most current systems, constraints in the goal and in rule bodies are evaluated sequentially, left-to-right, as formally specified in the refined operational semantics (Section 4.2.3). In CHR^{FP}, on the other hand, all constraint conjunctions are evaluated atomically, in batch—i.e., all constraints are added to their stores *before* the next rule is fired (cf. Section 4.4.2).

We believe batch evaluation is indeed a favourable language feature, for the following reasons:

1. It is indispensable for any non-trivial *execution control* mechanism that selects among applicable rule instances: all constraints in a body must be added to the store before selecting the next rule, precisely because these constraints determine the set of applicable rules
2. As shown in Chapter 6, batch semantics is similarly essential for a proper integration of other *language extensions* such as negation and aggregates.
3. Batch semantics add a natural symmetry to the *operational semantics* of the language. Before, a CHR rule only removed constraints in a single state transition. With batch semantics, the constraints in the body are added accordingly as well.
4. Batch semantics is a better match with CHR's *declarative* origins in logic. For pure constraints, the order in which they are added should not matter. Or, more precisely, the user should not (have to) specify this order, at least not by default. The compiler and runtime should instead decide the optimal evaluation order.

Nevertheless, left-to-right execution is familiar to most programmers, and many existing programs heavily rely on it. In fact, the following examples show that CHR's conventional sequential execution is often even preferable.

Example 5.8. The following example is given by De Koninck (2008):

```
a(X) <=> write('Give a number'), read(Y), Z is X - Y, b(Z).
```

Bodies containing side-effecting host language statements should obviously be executed left-to-right. Also, in this case, the value Y must be read before it can be used in an arithmetic computation. CHR^{FP}'s batch semantics does not guarantee this. De Koninck (2008) therefore proposes the following encoding:

```
3 :: a(X) <=> io(yes,write('Give a number'),Next),
           io(Next,read(Y),_), safe_is(Z,X - Y), b(Z).
1 :: io(yes,Call,Done) <=> call(Call), Done = yes.
1 :: safe_is(X,Y) <=> ground(Y) | X is Y.
```

Instantiation exceptions in arithmetic expressions are avoided by an appropriate guard. Left-to-right execution of IO operations is realised essentially by creating a chain of `io/3` constraints, linked together by free logical variables. Next, the `io/3` are triggered one by one by instantiating these link-variables. This pattern always works, but puts a heavy burden on the programmer.

Example 5.9. As a slightly more realistic example, consider the following CHR(Prolog) rule:

```
fact(N,F) <=> N1 is N-1, fact(N1,F1), F is F1 * N.
```

If this rule’s body is evaluated atomically, ‘`F is F1 * N`’ will raise an error because `F1` will be unbound. Again, auxiliary constructs such as `safe_is/2` in the previous example have to be used to mask the impure built-in constraint.

The latter example shows that simply executing built-in constraints from left-to-right, as pragmatically done by De Koninck (2008)’s CHR^{IP} implementation, is insufficient. CHR constraints can cause impure effects indirectly.

All examples so far were concerned with impure host-language built-ins. Even without such host-language interactions, however, the order in which CHR constraints themselves are evaluated often matters. Firstly, sequential order facilitates performance tuning. Secondly, constraints that represent operations or actions generally have to be evaluated sequentially. Such constraints are very common, particularly in (often non-confluent) general-purpose programs.

Example 5.10. A well-studied example is the union-find algorithm by Schrijvers and Frühwirth (2006), where the order of union and find operations clearly determines the result returned by find operations.

Example 5.11. Lam and Sulzmann (2006) demonstrated this issue using a simple blocks world CHR program with `get/1` and `put0n/2` actions. Their solution is based on a special type of CHR constraints and rules, called *action constraints* and *action rules*. They formulated a monadic action CHR semantics, which determines that action constraints are **Introduced** sequentially. Action rules were implicitly given the highest priority (without an explicit notion of priorities), and at most one action constraint was allowed in the store at all times (if no action rule is applicable, the computation blocks).

Our more pragmatic solution is far less restricted than that of Lam and Sulzmann (2006). CHR2 supports two types of conjunctions: *batch conjunction*, separated by ‘&’, and *sequential conjunction*, separated by ‘,’.⁵ Both types can be mixed freely (the ‘&’ operator has higher precedence than ‘,’).

Example 5.12. To illustrate this, consider the following CHR2(Prolog) program that implements a naive recursive computation of Fibonacci numbers:

⁵This way, CHR2 remains backwards compatible with ω_r -based systems.

```

fib(N,M), N=< 1 <=> M = 1.
fib(N,M), N > 1 <=>
  (N1 is N-1, fib(N1,M1)) & (N2 is N-2, fib(N2,M2)), M is M1+M2.

```

Intuitively, before the next conjunct of a sequential body conjunction is considered, all previous conjuncts are evaluated completely. That is, up to the same priority as the conjunction's rule, at least all newly applicable rules have fired exhaustively. In our example, this ensures that $M1$ and $M2$ are instantiated when the last conjunct is executed.

A batch semantics essentially only guarantees all its constraint conjuncts are **Introduced** and **Solved** before a next rule is allowed to fire. The order in which batch conjuncts are evaluated is not determined (they may even be evaluated in parallel). For instance, while both smaller Fibonacci numbers must be computed first, the order in which this happens is not important.

While intended to be intuitively clear, the complete, precise semantics of (nested) conjunctions is formally defined in Section 5.2.

Conclusion Even though batch conjunctions are more declarative, sequential conjunctions form a natural, useful control mechanism, orthogonal to rule priorities. Supporting both types facilitates both logical, declarative constraint conjunctions, as well as the selective introduction of sequentiality.

5.1.6 Set semantics

Often, constraints exhibit, or should exhibit, *set semantics* rather than CHR's default multiset semantics. The following example illustrates how set semantics may be crucial for a program's runtime complexity or termination behaviour.

Example 5.13. The EQ program is a typical CHR program. It nicely captures the three standard rules in the definition of an equivalence relation, modelled here as a binary `equiv/2` CHR constraint:

```

reflexivity @ equiv(X,X) <=> true.
symmetry @ equiv(X,Y) ==> equiv(Y,X).
transitivity @ equiv(X,Y), equiv(Y,Z) ==> equiv(X,Z).

```

Due to its `symmetry` rule, this program typically leads to infinite derivations of propagated duplicate constraints. Obviously, it needs a rule of form:

```

idempotence @ equiv(X,Y) \ equiv(X,Y) <=> true.

```

De Koninck (2008) indisputably demonstrated that under ω_r such *constraint store invariants* cannot be adequately enforced. Unfortunately, as noted also by De Koninck, priorities do not fully solve the problem either: giving this rule the

highest priority may not be enough to guarantee termination.⁶ The reason is that its semantics, ω_p , does not determine which of the two duplicate constraints is removed. If the wrong duplicate is consistently removed, infinite derivations remain possible. The same is true for ω_2 , the operational semantics of CHR2 we define later.

De Koninck (2008) therefore proposed an encoding where set semantics constraints are added in two phases. For our example (using CHR2 syntax):

```
check_duplicate @ -new_equiv(X,Y), +equiv(X,Y).
no_duplicate @ -new_equiv(X,Y) => equiv(X,Y).

priority check_duplicate > no_duplicate.
```

In the query and rule bodies, all `equiv` constraints moreover have to be replaced with `new_equiv`. Also, for non-ground CHR constraints the *idempotence* rule must be retained, as variable instantiations may introduce additional duplicate CHR constraints (as in ‘`new_equiv(X,Y), new_equiv(X,Z), Y = Z`’).

Such encodings are unnecessarily impractical and error-prone. De Koninck (2008) acknowledged this shortcoming, but neglected to work out an adequate solution. Set semantics, however, is prevailing in many CHR programs. In fact, CHR programs that actually use multiset constraints probably are more the exception than the rule. As discussed in Section 4.2, multiset semantics also matches poorly with CHR’s classical logical foundation. Similar observations were made e.g. by Betz et al. (2009), whose persistent constraints have set semantics, and Sarna-Starosta and Ramakrishnan (2007), whose CHRd system only offers set semantics constraints.

We propose that in CHR2 set semantics constraints can be declared as part of the CHR constraint declarations (nearly every system already requires CHR constraints to be declared explicitly). The precise syntax used is system-dependent. Using a CHR(Prolog)-style syntax, this gives for instance:

```
:- chr_constraint equiv/2 # set.
```

Each time a set semantics constraint is told, the constraint store is first checked for syntactically identical constraints. If present, the newly asserted constraint is silently removed. Duplicates caused by later unifications are resolved nondeterministically.⁷

5.1.7 Functional dependencies

A second important constraint invariant is the existence of *functional dependencies* (FDs) between constraint arguments. Functional dependencies are an established

⁶The *reflexivity* rule must also be given the highest priority, but that is not the point.

⁷This nondeterminism is kept after careful consideration. More fine grained control is always possible in CHR2 through explicit encoding (using e.g. explicit constraint identifiers).

concept in relational databases, and were first studied in detail for CHR by Duck (2005). We briefly introduce the basic notion using CHR rules:

Example 5.14. The following CHR rule expresses a functional dependency:

$$c(X, Y_1, _), c(X, Y_2, _) \implies Y_1 = Y_2.$$

The first argument is said to *functionally determine* the second: for all $c/3$ constraints with the same first argument, the second argument will be equal as well. Often, one or more arguments functionally determine all other arguments:

$$c(W, X, Y_1, Z_1), c(W, X, Y_2, Z_2) \implies Y_1 = Y_2, Z_1 = Z_2.$$

We call this a *full* functional dependency. In practice, constraints with full dependencies usually have set semantics as well:

$$c(W, X, Y_1, Z_1) \setminus c(W, X, Y_2, Z_2) \iff Y_1 = Y_2, Z_1 = Z_2.$$

Duck (2005) studies mainly the latter type, which he calls *set semantics functional dependencies*.

De Koninck (2008) showed that enforcing functional dependency invariants in CHR suffers from the same execution control issues as set semantics. This is especially true for set semantics functional dependencies:

Example 5.15. A more efficient CHR2 program to compute Fibonacci numbers than that of Example 5.12 might look like this:

```
table @ +fib(N,M1), -fib(N,M2) => M1 = M2.
priority table = highest.
+fib(N,M), N ≤ 1 => M = 1.
+fib(N,M), N > 1 => fib(N-1,M1) & fib(N-2,M2), M = M1 + M2.
```

The intention is that Fibonacci numbers are computed top-down, but results for lower values of N are reused instead of recomputed. This technique is called *memoisation* or *tabling*. For this to work as intended here, the *table* rule must remove the *fib/2* constraint that has not yet fired the propagation rule.

Analogous to the set semantics annotation, we therefore argue CHR2 systems should support concise functional dependency declarations. We briefly introduce a possible syntax by example.

Example 5.16. In JCHR2 (cf. Section 7.3), the constraints of the RAM handler could be declared as follows (note again the named constraint arguments):

```
public constraint pc(int label) # fd( --> label), // or singleton
mem(int addr, int val) # fd(addr --> val), // or key(addr)
prog(int label, int, int, int) # fd(label --> _); // key(label)
```

Here ‘-->’ denotes a set semantics FD (‘->’ is used for regular FDs). Shorthand notation ‘key’ for full set semantics FDs, and ‘singleton’ for *singleton* constraints (for which at most one instance exists), are also supported.

5.2 Operational Semantics

The main goal of the operational semantics of CHR2 is that it facilitates an effective, declarative programming style. On the one hand, it should stimulate the user to focus in the first place on the program's logic, by means of unordered logical CHR rules and constraint conjunctions. On the other hand, it must facilitate both intuitive and precise execution control. Any unintentional restrictions on the execution strategy must be avoided though at all costs.

For our definition of the operational semantics of CHR2 in Section 5.2.2, we assume a CHR2 program \mathcal{P} is first normalized by a series of source-to-source transformations, as outlined in Section 5.2.1.

5.2.1 Program normalisation

In a normalized program, denoted \mathcal{P}^* , each rule has the following normal form:

$$r :: p @ H, G \Rightarrow B_1, \dots, B_n$$

with H and G conjunctions of CHR and built-in constraints respectively. For rules with an empty head, $H = \$init$, with $\$init/0$ a special CHR constraint. All syntactic sugar, such as that of Section 5.1.3, is expanded.

All rule bodies are reduced to sequential conjunctions of batch conjunctions B_i that themselves no longer contain nested conjunctions. The transformation used when bodies are otherwise nested is best introduced by example:

Example 5.17. The rule in Example 5.12 is transformed into a set of rules of the following form. Note that we apply the common convention that compiler-generated names start with a '\$' symbol:

```
$rule1 @ fib(N,M) <=> $aux1(N,M1) & $aux2(N,M2), M is M1+M2.
$aux($rule1,1) @ $aux1(N,M1) <=> N1 is N-1, fib(N1,M1).
$aux($rule1,2) @ $aux2(N,M2) <=> N2 is N-2, fib(N2,M2).

priority $aux(X,_) > X.
```

Each sequential conjunction nested inside a batch conjunct is thus encoded as a single CHR constraint, which is expanded back to its original form by a corresponding auxiliary rule. By exhaustively applying this transformation, all rule bodies are reduced to the required normal form.

Set semantics and functional dependency invariants could in principle be incorporated in the **Introduce** and **Solve** transitions of the semantics (see later). To simplify the semantics and its discussion though, we again opted to employ a straightforward transformational approach. The required auxiliary constructs have been discussed in Sections 5.1.6 and 5.1.7.

Lastly, all constraint removals are made explicit as conjuncts of the first batch B_1 using the **remove/1** operation defined in Section 5.1.2. Each rule is assigned a unique identifier ρ implicitly (required for the propagation history).

5.2.2 The operational semantics ω_2

The operational semantics ω_2 defines the semantics of any normalized CHR2 program \mathcal{P}^* as a state-transition system.

Definition 5.1 (Action). In a body conjunction (either sequential or batch), an *action* is either a CHR or built-in constraint, or a **remove**(*id*) operation. The former type of action is called an *addition*, the latter a *removal*.

Definition 5.2 (Execution state). An ω_2 *state* is a tuple $\langle \mathbb{G}, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$, with \mathbb{S} , \mathbb{B} , \mathbb{T} , and n defined in Definition 4.2 (page 38). The goal \mathbb{G} is a set of actions, and the stack \mathbb{A} is a sequence of tuples (G, p, A) with G a sequence of sets of actions (a sequential conjunction of batch conjunctions), p an instantiated rule descriptor that determines the priority of these actions, and A the set of applicable rule instances from right before the previous batch was evaluated. We further define the projection operator $\text{prio}(\mathbb{A}) = \{p \mid \exists(G, p, A) \in \mathbb{A}\}$.

Given an initial query Q , i.e. a sequence of batches of constraints, the initial state is $\langle \emptyset, [(Q', \text{lowest}, \emptyset)], \emptyset, \text{true}, \emptyset \rangle_1$. In Q' , the first batch of Q is extended with **\$init**, to accommodate normalized headless rules (see Section 5.2.1).

Figure 5.1 shows the transitions of the ω_2 semantics. The first three transitions execute all actions of a given batch conjunction. The **Solve** and **Introduce** transitions are identical to their counterparts in ω_t and ω_p (Figure 4.1); the **Remove** transition is a straightforward extension to deal with (explicit) removals. As in ω_p (Figure 4.3), batch semantics is enforced by requiring the goal to be empty before any of the last three transitions applies.

The **Apply** transition is also similar to that of ω_p , in the sense that no rules with a higher priority may be applicable. With the set of matching substitutions $\text{matchings}(H, G, S, \mathbb{B})$ defined analogously to Definition 4.3, the set of applicable rule instances is defined as follows:

Definition 5.3. For a set of instantiated rule descriptors P , the set of *applicable rule instances*, denoted $\text{applicable}(P, \mathbb{S}, \mathbb{B}, \mathbb{T})$, consists of those *rule instances* $(\rho :: \theta(p), S)$, for which ρ is a rule of form $\rho :: p @ H, G \Rightarrow B_1, \dots, B_m$, $S \subseteq \mathbb{S}$, $\theta \in \text{matchings}(H, G, S, \mathbb{B})$, $(\rho, \text{ID}(S)) \notin \mathbb{T}$, and $\forall p' \in P : \mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \theta(p) \not\prec p'$.

As explained next, the **Apply** transition contains some additional conditions though related to sequential conjunctions.

1. **Solve** $\langle \{b\} \uplus \mathbb{G}, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}^*} \langle \mathbb{G}, \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$
where b is a built-in constraint.
2. **Introduce** $\langle \{c\} \uplus \mathbb{G}, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}^*} \langle \mathbb{G}, \mathbb{A}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
where c is a CHR constraint.
3. **Remove** $\langle \{remove(id)\} \uplus \mathbb{G}, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}^*} \langle \mathbb{G}, \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \rangle_n$
with $\mathbb{S}' = \{c\#i \in \mathbb{S} \mid i \neq id\}$ (if not already removed, $c\#id$ is removed).
4. **Apply** $\langle \emptyset, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}^*} \langle B_1, [([B_2, \dots, B_m], \theta(p), A)|\mathbb{A}], \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T}' \rangle_n$
where $\rho :: p @ H, G \Rightarrow B_1, \dots, B_m$ is a renamed apart rule of \mathcal{P} for which $S \subseteq \mathbb{S}$, $\theta \in \text{matchings}(H, G, S, \mathbb{B})$, $\mathbb{T}' = \mathbb{T} \sqcup (\rho, \text{ID}(S))$, and $\forall (\rho' :: \theta'(p'), S') \in \text{applicable}(prio(\mathbb{A}), \mathbb{S}, \mathbb{B}, \mathbb{T}) : \theta'(p') \neq \theta(p)$. Moreover, $A = \text{applicable}(\{\theta(p)\} \cup prio(\mathbb{A}), \mathbb{S}, \mathbb{B}, \mathbb{T})$ and $\forall p' \in prio(\mathbb{A}) : p' \neq \theta(p)$.
5. **Batch** $\langle \emptyset, [([B|Bs], p, A)|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}^*} \langle B, [(Bs, p, A')|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
with $A' = \text{applicable}(\{p\} \cup prio(\mathbb{A}), \mathbb{S}, \mathbb{B}, \mathbb{T})$ such that $A' \subseteq A$.
6. **Pop** $\langle \emptyset, [([], p, A)|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}^*} \langle \emptyset, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
The **Pop** transition is eagerly applied whenever possible.

Figure 5.1: The transition rules of the operational semantics ω_2 . All transitions have as an additional, implicit precondition that $\mathcal{D}_{\mathcal{H}} \models \exists_{\emptyset} \mathbb{B}$.

Sequential conjunction

The precise semantics of sequential conjunction warrants further discussion. The intuition is that a sequential conjunct is only evaluated after the previous conjunct is ‘fully evaluated’. That is, all rules ‘caused’ by adding the previous batch must have been fired. The **Batch** transition contains our attempt at formalising this intuition: the next batch may only be evaluated when all rule instances that are applicable were already applicable before the previous batch was evaluated. Moreover, in order to obtain intuitive priority invariants, as long as a rule of some priority p is not fully evaluated, no rules of priority lower than p are allowed to fire. This is enforced by the last condition in the **Apply** transition (the same condition also occurs in Definition 5.3).⁸

5.2.3 Compatibility with other semantics

In this Section, we formally verify the compatibility of ω_2 semantics with all relevant existing CHR semantics.

Compatibility with ω_t

We define an abstraction function α_2 from ω_2 states and derivations to their ω_t counterparts. Define $removals(\mathbb{G}) = \{x \in \mathbb{G} \mid x \text{ is of form } remove(id)\}$. Then, the abstraction of states is defined as follows:

$$\alpha_2(\langle \mathbb{G}, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{G} \cup \left[\bigcup_{(G,p,A) \in \mathbb{A}} \left(\bigcup_{B \in G} (B \setminus removals(B)) \right) \right], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$$

The abstraction of derivations though is slightly complicated by **Remove** transitions. First, we should only consider programs that, prior to normalisation, do not contain any explicit $remove(id)$ actions. For non-propagation rules, however, normalisation does add a number of removal actions to the first batch of the body—one for each removed occurrence. In ω_2 , **Solve** and **Introduce** transitions are allowed to occur prior to the **Remove** transitions that remove the removed occurrences. In ω_t , this is not possible.

However, neither the order in which removals occur, nor their order with respect to addition actions, has any effect on the the first **Batch**, **Pop**, or **Apply** transition that follows. We therefore first define an auxiliary abstraction function that essentially performs all removals in a state’s goal atomically throughout the entire remaining derivation:

⁸In (Van Weert et al. 2009), we presented an earlier, incorrect version of the ω_2 semantics. It was flawed in several ways, not in the least because it allowed these natural ω_p -like priority invariants to be broken.

Definition 5.4. Let $\sigma = \langle B, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ and $R = \{c\#id \in \mathbb{S} \mid \text{remove}(id) \in B\}$. Then $\beta_\sigma([\langle G, \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n | D']) = [\langle G, \mathbb{A}, \mathbb{S} \setminus R, \mathbb{B}, \mathbb{T} \rangle_n | \beta_\sigma(D')]$.

Using this, we extend α_2 to arbitrary ω_2 derivations D . For the trivial derivation $\alpha_2([\sigma]) = [\alpha_2(\sigma)]$. Otherwise $D = [\sigma_0, \sigma_1 | D']$ starts with a transition $\sigma_0 \rightsquigarrow \sigma_1$. In case this transition is an **Apply** transition, we define $\alpha_2(D) = [\alpha_2(\sigma_0) | \alpha_2(\beta_{\sigma_1}([\sigma_1 | D']))]$. In all other cases, α_2 is defined analogously to the α abstraction function used in Section 4.2.3 for ω_r :

$$\alpha_2(D) = \begin{cases} \alpha_2([\sigma_1 | D']) & \text{if } \alpha_2(\sigma_0) = \alpha_2(\sigma_1) \\ [\alpha_2(\sigma_0) | \alpha_2([\sigma_1 | D'])] & \text{otherwise} \end{cases}$$

Theorem 5.1 (ω_t correctness). *Suppose the program \mathcal{P} does not contain any explicit removal actions. Let \mathcal{P}' be equal to \mathcal{P}^* , only without the explicitized removals (i.e. all nested conjunctions are still normalised). Then for every \mathcal{P}^* -derivation D under ω_2 , a corresponding \mathcal{P}' -derivation $\alpha_2(D)$ exists under ω_t . If a state σ_f is a final state under ω_2 , then $\alpha_2(D)$ is a final state under ω_t .*

Proof sketch. Correctness holds because, compared to ω_t , ω_2 only adds restrictions on the order of **Apply**, **Solve** and **Introduce** transitions. The full proof is by induction on derivation length. \square

In fact, the theorem still holds without the normalisation to \mathcal{P}' , but then the abstraction function must be extended to further collapse all **Apply** transitions corresponding to generated rules.

Theorem 5.2 (ω_t completeness). *Let \mathcal{P} be a program without explicit removals or sequential conjunctions, and whose priority constraints imply all rule instances either have equal or incomparable priority. Then for every \mathcal{P} -derivation D under ω_t , a corresponding \mathcal{P}^* -derivation D' exists under ω_2 for which $\alpha_2(D') = D$.*

Proof sketch. The derivation D' can easily be constructed from D not removing CHR constraints from the store in **Apply** transitions, but instead adding the obvious $\text{remove}(id)$ actions to the goal, and introducing the necessary **Remove** transitions after each **Apply**. Moreover, a correct stack argument \mathbb{A} is easily added to each state. Because all rule bodies consist of a single batch conjunction, after every **Apply** transition, the stack contains one single element (containing an empty sequence of batches). Once all actions of the rule's single batch are evaluated, a **Pop** transition must always be added to D' by definition. Consequently, the stack \mathbb{A} will never contain more than one (trivial) tuple, D' will never contain a **Batch** transition, and in the starting state of every **Apply** transition in D' , the stack will be empty. In other words, the stack never influences rule applicability. Therefore, the derivation D' thus constructed is a valid ω_2 derivation. \square

As a consequence of Theorem 5.1 the soundness and completeness results for CHR programs with respect to their different logical semantics discussed in Section 4.2.2 still hold for ω_2 . Theorem 5.2 implies that for CHR2 programs with batch conjunctions only, and without (real) priority constraints, every execution strategy under ω_t (including ω_r) is consistent with ω_2 .

In the next two subsections, we obtain even stronger compatibility results with specific instances of ω_t .

Compatibility with ω_p

Any CHR^{FP} program (where all bodies are batch conjunctions) can be turned into an equivalent CHR2 program by adding priority constraints such as those defined in Example 5.7. The reverse is not always true: not every CHR2 program can readily be executed under CHR^{FP}. A set of CHR2 rule descriptors may form a partially (or even totally) ordered set that is not locally finite, that is, there may be unbounded intervals of rule descriptors. We have given a concrete counterexample earlier in Example 5.6. Any CHR2 program with static rule priorities only though can be trivially transformed into a CHR^{FP} program.

Let \mathcal{P} be any CHR^{FP} program, and \mathcal{P}^* the same program normalized and augmented with priority constraints as in Example 5.7. Then, with α_2 defined as before, the following strong equivalence theorems can be proven:

Theorem 5.3 (ω_p correctness). *Let D be any \mathcal{P}^* -derivation under ω_2 , whose initial state contains only one batch sequence. Then there exists a corresponding \mathcal{P} -derivation $\alpha_2(D)$ under ω_p . If a state σ_f is a final state under ω_2 , then it $\alpha_2(D)$ is a final state under ω_p .*

Proof. By induction on the length of the derivation. □

Theorem 5.4 (ω_p completeness). *For every \mathcal{P} -derivation D under ω_p , a corresponding \mathcal{P}^* -derivation D' exists under ω_2 for which $\alpha_2(D') = D$.*

Proof. Analogous to Theorem 5.2. □

When running a CHR^{FP} program under ω_2 , all results obtained are correct (Theorem 5.3) and no possible outcome is lost (Theorem 5.4). All results by De Koninck (2008) on confluence and complexity of CHR^{FP} programs therefore still apply for this subset of CHR2 programs. Extending these results to more general CHR2 programs is possible as well.

Conversely, let \mathcal{P} be any CHR2 program that can be transformed into a CHR^{FP} program \mathcal{P}' (see earlier), e.g. by computing a linear extension of the partial order implied by the priority constraints. Then Theorem 5.4 still holds for \mathcal{P}' and \mathcal{P} —i.e., such programs can correctly be executed using CHR^{FP}—but Theorem 5.3 does not: in general, CHR2 allows derivations that are not allowed

by CHR^{TP} —i.e., turning the partial order into a total order adds (unintentional) restrictions on the execution order.

Compatibility with ω_r

Let \mathcal{P} be any CHR program with sequential conjunction only, and \mathcal{P}^* its normalized version augmented with priority constraints implying that all rules have equal priority. Then, with α_2 defined as before, and α as in Section 4.2.3:

Theorem 5.5 (ω_r completeness). *For every \mathcal{P} -derivation D under ω_r , there exists a corresponding \mathcal{P}^* -derivation D' under ω_2 for which $\alpha_2(D') = \alpha(D)$.*

Proof sketch. In ω_r , adding a constraint c in a **Introduce** and **Solve** transition is always followed by one or more constraint (re)activations. These, by construction, recursively and exhaustively fire all rules that became applicable by adding c .⁹ In other words, the condition in the **Batch** transition will always be fulfilled when constructing D' . \square

In other words, ω_r correctly enforces sequential conjunction, and any ω_r -compliant system can be used to execute existing programs.

The obvious correctness theorem—similar to Theorems 5.1 and 5.3—does not hold. That is: by design, ω_2 allows derivations that are not necessarily allowed under ω_r . This stems from two reasons:

1. The first and most obvious reason is that ω_r forces rules to be executed in textual order.
2. A second reason is the nondeterminism in the ω_2 semantics of sequential conjunction. While, after each sequential conjunct, ω_2 enforces the exhaustive application of newly applicable rule instances, it purposely allows rule applications that were already applicable earlier as well. In principle, any applicable rule instance found during the evaluation of a sequential conjunct should be allowed to fire, even if it was already applicable before. It is not even clear in general how this could (reasonably) be avoided. In fact, in ω_r , the **Solve** transition is nondeterministic for analogous reasons: see Section 4.2.3. Nevertheless, ω_r specifies more precisely the set of rule instances that may fire after each sequential conjunct (and not only their order).

In (Van Weert et al. 2009), we formulated a refined instance of ω_2 , that, when used to evaluate legacy CHR programs, precisely reduces to the ω_r semantics.¹⁰

⁹Due to the underdetermined **Solve** transition, rule instances that were already applicable prior to adding c may also fire. This is allowed though by the ω_2 semantics.

¹⁰The ω_r^2 semantics of (Van Weert et al. 2009) is incorrect when it comes to dealing with priorities, but this could easily be fixed.

Such a semantics could constitute the basis for fully backwards compatible CHR2 systems. For a while, it was implemented by JCHR2 (Section 7.3), until it became clear that it prohibited certain rule application reorderings a compiler should be allowed to make. While an optional ‘backwards compatible’ mode might remain useful, we purposely do not formulate any refined semantics here. As discussed in the next section, new CHR2 programs should in principle only rely on what is determined by the ω_2 semantics.

5.2.4 Discussion

The ω_2 semantics is specifically crafted to be as nondeterministic as possible, while still fixing the essential semantical properties of CHR2’s basic building blocks introduced in Section 5.1. Obviously, it is difficult to validate ω_2 ’s suitability or utility. In Section 5.2.3, we verified a first prerequisite, namely the compatibility with established, related operational semantics. The real question though is whether it strikes the right level of determinism, facilitating a naturally controllable runtime behaviour, but without abolishing CHR’s high-level, logical, declarative nature.

We believe a good way to characterize a good CHR operational semantics is: “*a theoretical semantics that needs no refining*”, or “*an ω_t that needs no ω_r* ”. The ω_r semantics was introduced because ω_t , while it has nice declarative properties, was found to nondeterministic for practical programming. The nearly deterministic ω_r semantics though requires users to reason on a very specific execution mechanism, completely in contrast to the declarative ‘what, not how’ motto. In fact, users *always* almost completely fix the execution strategy, even if they do not intend it. An arbitrarily chosen order of sequential conjuncts or rules still determines the evaluation order. A normative ω_r semantics therefore demeans CHR’s logical, declarative roots, injecting several low-level, procedural, imperative characteristics into its semantics.

CHR^P and ω_p were designed towards a similar goal, but are still not free of similar determinism-related issues: they determine a total, global order on rule instances, yet lack the sequential control required by many practical programs.

Unlike De Koninck (2008), we explicitly do not formulate a refined operational semantics for CHR2. As already pointed out at the end of the previous section, once users start to rely on it, any such semantics would prohibit future improvements in execution strategies. Well-written CHR2 programs should only rely on what is determined by the ω_2 semantics. Of course, only hands-on experience can truly determine whether this is sufficient for practical programming. We are therefore developing a reference implementation of CHR2(Java). This system, called JCHR2, is discussed in more detail in later chapters.

5.3 Conclusions

In this chapter, we outlined the basic building blocks of a next generation CHR language called CHR2. For the first time since the conception of the CHR language, we propose to modernise the syntax of rules. We have clearly shown that the streamlined, backwards compatible CHR2 syntax allows for more natural, readable, and concise rule definitions. We furthermore introduced compact syntax for the declaration of common constraint invariants, invaluable both as program documentation and for advanced compiler optimisations.

Compared to current generation ω_r -based CHR systems, the operational semantics ω_2 of CHR2 programs is considerably less deterministic. The order in which rules are defined is no longer significant, and by default logical, more declarative batch conjunctions are used. Precise execution control can be added effectively and intuitively in the form of sequential conjunctions and rule priorities. Unlike before, sequentiality is optional, and priority constraints allow for local specifications of rule priorities. Completely in line with the declarative ideal, any redundant constraints on the execution strategy are avoided. Essentially, ω_2 is a first theoretical operational semantics that needs no refining, a semantics that is very high-level and declarative, yet still facilitates the effective execution control required for practical programming. We have proven that ω_2 is fully compatible with all relevant operational semantics of CHR.

The CHR2 language is designed to be a solid, extensible basis for additional language features. In Chapter 6, for instance, we show that—unlike traditional ω_r -based systems—CHR2 is ideally suited to host aggregates. As with any high-level declarative language, CHR2 offers a lot of freedom to the compiler. The challenging problem of efficiently executing CHR2 programs is discussed in detail in Part III. A reference implementation of CHR2(Java), JCHR2, is introduced in Section 7.3.

Chapter 6

Aggregates

A programming language is low level when its programs require attention to the irrelevant.

— **Alan Perlis** (1922–1990)
American computer scientist

A frequently recurring class of programming idioms where CHR’s conciseness and expressiveness is lacking is the aggregation of information from nontrivial, possibly unbounded parts of the constraint store. Examples include counting the number of constraints that meet certain conditions, and finding the minimal value for some constraint’s argument. Each individual CHR rule only considers a fixed, bounded number of constraints, equal to the number of conjuncts in its head. Aggregations therefore always require an explicit encoding using multiple auxiliary rules and constraints. Such ad hoc approaches are repetitive, cumbersome and error-prone, and the resulting auxiliary constructs tend to cross-cut the entire program. This severely handicaps all advantages of declarative programming, such as conciseness, readability and maintainability.

We propose an extension of CHR with *aggregates*, in which heads may contain aggregate conditions ranging over CHR constraints. With language support for aggregates the programmer can express the aggregate logic concisely in single self-contained rules, exhibiting all advantages of declarative programming.

Overview In Section 6.1, we use illustrative examples to further motivate the need for proper aggregation abstractions. Next, we introduce a powerful, extensible aggregate framework, first informally in Sections 6.2 and 6.3, and then more formally in Section 6.4. Using four case studies, Section 6.5 clearly demonstrates the added expressiveness. Finally, Section 6.6 compares with related

work, and Section 6.7 concludes and offers some prospects for future research.

We implemented both negation as absence and aggregates for CHR(SWI-Prolog) using source-to-source transformations to optimised, regular CHR rules. These contributions are presented in detail in Van Weert, Sneyers, et al. (2006b, 2008). This chapter focusses solely on language design aspects.

6.1 Motivation

As CHR is Turing complete (see Section 4.3.3), no language extension can add computational power. Nevertheless, we will show that aggregates are invaluable when it comes to expressiveness, maintainability and conciseness.

We start in Section 6.1.1 by discussing *negation as absence*. The need for this specific aggregate has been particularly felt by CHR programmers. We will show that explicitly encoding even this deceptively simple aggregate is cumbersome and non-trivial. In Section 6.1.2, we illustrate these issues carry over to other, more general aggregate functions.

6.1.1 Negation as absence

The traditional interpretation of CHR simplification in the declarative specification of constraint solvers is that CHR constraints are not so much *removed* as they are *rewritten* to a simpler, more canonical form—hence the term *simplification*. This is also reflected in the logical interpretation of CHR simplification as logical equivalence (Section 4.2.1).

CHR, however, can also be viewed as a forward chaining, data-driven language, much like production rules (augmented with elements of CLP languages). It is in this respect that CHR is increasingly being used as a general programming language, in a wide range of applications. In fact, CHR implementations of more advanced constraint solvers typically also apply CHR this way (e.g. INCLP(\mathbb{R}) by De Koninck et al. (2006a); cf. Section 4.6.1). In these cases, CHR ‘constraints’ commonly represent procedural data, such as flags, locks, or other execution control constructs, or elements in some data structure, resource handlers, properties, etc. Often, the lack or removal of such information is perceived as meaningful. The CHR language though offers no constructs to test the *absence* of CHR constraints, or to react to their *removal*.

Example 6.1. Already more than a decade ago, CHR programmers have felt the need for negation as absence. In 1997, Christian Holzbaaur wrote a CHR port of a classical production rules program that solves (McCarthy 1963)’s famous ‘*Monkey and Banana*’ toy AI planning problem. In 9 out of the 25 rules, the original program tested for the absence of constraints (recall from Chapter 2 that negation as absence is an established feature in production rule languages).

Holzbaur therefore wrote a 'not'/1 auxiliary predicate to use in the guard of the ported rules. The following lines are taken verbatim from his program¹:

```
:- op(900,fy,not).
% There is no such fact ('not exists' in SQL)
not Fact :- find_constraint( Fact, _), !, fail.
not _.
```

Note the fitting comment by Holzbaur, as well as the striking analogy with the definition of negation as failure in Example 3.8 on page 25.

Unfortunately, while concise and readable, this approach is unportable and particularly inefficient. The reason is that the `find_constraint/2` built-in of the now deprecated CHR(SICStus) system linearly traverses (using backtracking) *all* constraints in the CHR constraint store.

Common approach: explicitly encoding absence tests

Over the years, several patterns to encode negation in CHR itself have become popular among CHR practitioners.

Example 6.2. We illustrate these idioms using the following CHR rule:

```
person(X), ~married(X) ==> single(X).
```

It uses a negation as absence extension of CHR—written using a prefix ‘~’ modifier—to express that: “If a person X is not married, then that person is single.” In regular CHR, this can be encoded in at least the following ways (see also Frühwirth 2009, §6.1.2):

1. **CHR constraints in the guard** While in theory CHR only allows built-in constraints in the guard, in practise calling a CHR constraint from the guard does work in most CHR(Prolog) systems. This is frequently exploited to explicitly encode absence tests as follows:

```
person(X) ==> not_married(X) | single(X).
married(X) \ not_married(X) <=> fail.
not_married(_) <=> true.
```

2. **Two-phase commit** This second pattern consists of replacing the body with an auxiliary constraint, and only applying the body if no constraints matching the negated heads are found:

```
person(X) ==> maybe_single(X).
married(X) \ maybe_single(X) <=> true.
maybe_single(X) <=> single(X).
```

¹As found on the CHR Website (2010).

This idiom works in any system that implements the refined operational semantics (or supports execution control by other means). The disadvantage is that it is only effective in case of propagation rules².

- 3. Provisional commit** A slightly shorter variant is to commit provisionally, immediately retracting the added constraints again if necessary:

```
person(X) ==> single(X).
married(X) \ single(X) <=> true.
```

Since built-in constraints can generally not be retracted, this pattern is only applicable for CHR constraints. It is also quite error-prone: if the `single/1` is not removed immediately, other rules may erroneously match with it. Under the ω_r semantics this can mostly be accomplished by putting the necessary simpagation rules first.³ The result though is that program logic becomes scattered, severely hampering readability and maintainability.

Example 6.3. A real-life instance of the ‘two-phase commit’ idiom introduced in Example 6.2 is found in the CHR(Prolog) implementation of Dijkstra’s algorithm by Sneyers et al. (2006a):

```
dist(N,L), edge(N,N2,W) ==> L2 is L+W, relabel(N2,L2).
dist(N,_) \ relabel(N,_) <=> true.
relabel(N,L) <=> doi(N,L).
```

Using negation, these three rules can be written as one single rule, eliminating the need for the auxiliary constraint `relabel/2` and the dependency on the execution order of the refined semantics:

```
dist(N,L), edge(N,N2,W), ~dist(N2,_) ==> L2 is L+W, doi(N,L2).
```

Unlike Holzbaaur’s approach, these encodings work in most CHR systems, and allow the CHR compiler to exploit available index structures. They do, however, inherently rely on (procedural) execution control mechanisms, and are quite verbose (typically one extra constraint and two extra rules per negation).

Reacting to removal

All approaches seen so far only test for the absence of certain constraints. This, however, only covers the passive aspect of negation as absence. The active aspect of negation entails reacting to constraint removal, that is, rules are expected to fire after constraints matching negated conditions are removed.

²CHR \mathcal{R} facilitates effective encodings of this pattern for other rules as well through the explicit removal operation of Section 5.1.2.

³As nicely shown by De Koninck (2008) though, such patterns unfortunately do not always work in the non-ground case, nor when more than one of these rules are required.

Example 6.4. A well-known CHR pattern to maintain the minimal element of a collection consists of the following two rules:

```
c(X) ==> min(X).
min(X) \ min(Y) <=> X ≤ Y | true.
```

Unfortunately these rules cannot consistently keep the minimum if elements are removed. In (Van Weert et al. 2006b), we generalised this pattern as follows (using CHR2 syntax and execution control here):

```
invalid_min @ -min(X), ~c(X).
new_min     @ +c(X), ~(c(Y), Y < X) => min(X).
filter_min  @ +min(X), -min(Y), X ≤ Y.
priority filter_min = highest, invalid_min > new_min.
```

When a minimal element $c(X)$ is removed, the semantics of negation dictates that the `invalid_min` rule must react to this, and remove the corresponding `min(X)` constraint. Similarly, the `new_min` rule must add a new, correct minimum. The priorities are required to make everything happen in the right order.

Implementing the above example in plain CHR requires cross-cutting changes to all rules that remove a `c/1` constraint, each time ensuring the minimum is correctly updated when needed. Using negation as absence, reacting to removal becomes manageable.

Nonetheless, the pattern of Example 6.4 remains quite verbose, and dependent on subtle execution control issues. More powerful language abstractions are therefore indispensable. In the case of Example 6.4, retrieving and maintaining a minimum should be provided by a suited `min` aggregate.

6.1.2 Aggregates

Example 6.5. Suppose that the two CHR constraints `client(ClientId)` and `account(AccountId,ClientId,Balance)` constitute a (simplified) representation of the clients and accounts of a bank. Initially, one of the business rules of the bank states:

“A platinum client is a client whose account balance is \$25,000 or more.”

This is readily expressed using the following CHR rule:

```
client(C), account(_,C,B) ⇒ B ≥ 25000 | platinum(C).
```

Clients, however, are allowed to have multiple accounts, so at some point the bank manager prefers to change the bank’s rule to:

“ A platinum client is a client whose accumulated sum of account balances is \$25,000 or more. ”

Unfortunately, expressing this in CHR is no longer straightforward.

We now compare three different approaches to implement the extended business rule of Example 6.5. The first two are possible in current CHR systems, whereas the third one uses aggregates.

Naive approach

If the maximum number of accounts per client is limited to some fixed number n , all possible cases are expressed in CHR as:

```
client(C), account(_,C,B) ==> B >= 25000 | platinum(C).
...
client(C), account(_,C,B1), ..., account(_,C,Bn)
==> B1+...+Bn >= 25000 | platinum(C).
```

Software engineering methodology dictates that the above replication of code is highly undesirable: it is hard to read and hard to maintain. This approach also scales very badly performance-wise, as the number of combinations tried during matching increases exponentially. Moreover, exhaustively enumerating all possible cases is clearly impossible if n is unbounded.

Common approach

A more concise solution, commonly used by CHR practitioners, is to introduce an auxiliary constraint `acc_balance/2`:

```
client(C), acc_balance(C,Sum) ==> Sum > 25000 | platinum(C).
```

This concisely captures the logic of platinum clients in a single rule. A second advantage over the naive approach is that it facilitates an unbounded number of accounts per client. This approach remains, nevertheless, inadequate, because it necessitates the maintenance of the accumulated balance. This inherently is a cross-cutting concern, as it requires invasive modifications to all parts of the original code that alter the balance of an account:

```
deposit(A,X), account(A,C,B) <=> account(A,C,B+X).
...
withdraw(A,X), account(A,C,B) <=> B > X, account(A,C,B-X).
```

All these rules, spread throughout the entire program, have to be adjusted to update `accumulated_balance` accordingly:

```
deposit(A,X), account(A,C,B), acc_balance(C,Acc) <=>
    account(A,C,B+X), acc_balance(C,Acc+X).
```

```

...
withdraw(A,X), account(A,C,B), acc_balance(C,Acc) <=>
    B > X, account(A,C,B-X), acc_balance(C,Acc-X).

```

Also, the accumulated balance has to be initialised for new clients:

```
client(C) ==> acc_balance(C,0).
```

Several variations to the above maintenance scheme can and have been concocted, but they all require similar modifications scattered throughout the entire program. As a result, this approach also displays poor compliance with common software quality criteria: it is very error-prone, and it impairs the readability and maintainability of the program, as the logic of many rules becomes tangled with obfuscating auxiliary code.

Aggregates

The use of aggregates shares the benefits of the previous approach, whilst dispensing with its drawbacks. Using an aggregate condition (in italics), the platinum client business rule is again declaratively expressed in a single rule, independent of the number of accounts:

```
client(C), sum(B, account(C,_,B), Sum)
    ==> Sum > 25000 | platinum(C).
```

No further changes to the program are required. A perfectly correct behaviour is already guaranteed implicitly by the aggregate's semantics.

As a result, the program is more declarative, readable and maintainable. The programmer's productivity is improved, because he is relieved from the cumbersome and repetitive task of implementing aggregates, and can entirely focus on his application domain.

6.2 Extensible Aggregate Framework

We have developed a generic framework for the specification of aggregates for rule-based languages. It is built on top of a universal aggregate language construct, powerful enough to express any aggregate function (Section 6.2.1). Not only can numerous predefined aggregates be supported directly by transforming them to this general construct, our approach also facilitates the specification of application-tailored aggregates by end users (Section 6.2.2).

Once we have established how aggregates can be specified, several more design issues arise when embedding them into CHR rules. These are discussed in Section 6.3. For now, it suffices to know that aggregates are extra rule applicability conditions, written at the left-hand side of CHR rules. Recall from Section 5.1.1 that, by design, CHR2's syntax readily allows such extensions.

6.2.1 Universal aggregate construct

An *aggregate* is essentially a function that returns a single value—albeit possibly a list or a set—computed from some specific subset of the CHR constraint store, without modifying the store. In this section we define a generic, high-level *universal aggregate construct*. Any aggregate can be formulated in terms of this construct (cf. Sneyers et al. 2007, §4.4 for a classification of expressible aggregates in terms of required computational resources).

To allow system implementors and end users to easily and effectively implement new aggregates, we believe the concrete format and syntax used must be host language dependent. Throughout this chapter, we will use Prolog as a host-language, and adopt the syntax used by our CHR(Prolog) reference implementation (Van Weert et al. 2008). We occasionally hint at possible equivalent constructs for other host languages.

In our CHR(Prolog) framework, the universal aggregate construct is:

```
aggregate(Init, Inc, Dec, Final, Element, Goal, Result)
```

Briefly, the function of the seven arguments is as follows:

Init	a predicate that returns the initial working value;
Inc	a predicate that takes the current working value and an element, and returns a new incremented working value;
Dec	similar, but returns a decremented working value (optional);
Final	a predicate that takes a working value and returns the result;
Element	a template to describe an element for a given Goal ;
Goal	a conjunction of CHR constraints, guards, and aggregates;
Result	returns the result of the aggregate.

The first four arguments determine the different procedures used to compute an aggregate. In our case, these are Prolog terms that determine which predicates must be called (after extending these terms with some extra arguments). In CHR(C), however, function pointers could for instance be used, whereas a typical CHR(Java) implementation would require a class to be specified, whose objects implement some specific interface that exposes the corresponding methods. Alternatively, an implementation could support syntax to define these four procedures directly in the CHR source code itself.

The last three arguments are analogous to the arguments of the well-known `findall/3` Prolog predicate discussed in Section 3.1.3 on page 25. Again, for other host languages, other variants may be more appropriate.

Example 6.6. The aggregate in Example 6.1.2 is equivalent to:

```
client(C), aggregate(=(0),plus,minus,=,B,account(C,_,B),Sum)
==> Sum > 25000 | platinum(C).
```

Here, the term ‘=’ denotes Prolog’s built-in ‘=’/2 unification predicate, and ‘=(0)’ unification with zero. The auxiliary predicates `plus/3` and `minus/3` are trivially defined as:

```
plus(X,Y,Z) :- Z is X + Y.    minus(X,Y,Z) :- Z is X - Y.
```

Operational semantics (informal)

Using Example 6.6 as a running example, we now illustrate the basic operational principles of *how* and *when* an aggregate value is computed:

How The working value W_0 of the aggregate is initialised by calling `Init(W_0)`⁴. In the case of `sum`, this results in an initial working value $W_0 = 0$. This value is then incremented once for each of the n matchings of `Goal`, by invoking `Inc(W_{i-1} , Element, W_i)` for $1 \leq i \leq n$. The order in which the matches are found is undetermined. For `sum`, the predicate `plus/3` increments the working value with the value of `Element` (`B` in the example). Finally, the last working value W_n is finalised by calling `Final(W_n , Result)`. Often, like for `sum`, this simply unifies the last working value with `Result`⁵. The aggregate’s computation, and in particular the finaliser predicate, is allowed to fail (in the Prolog sense). In that case, the aggregate is undefined and the rule containing it is not applicable. For instance, `max` or `avg` aggregates may fail when there are no elements.

The optional decrement predicate `Dec` is only used for an alternative computation strategy, where aggregate values are maintained incrementally. This is often required to obtain the correct runtime complexity. For more information on these crucial implementation aspects, we refer to (Van Weert et al. 2008).

When Operationally, a rule containing an aggregate is tried when one of the other head constraints is activated or reactivated; we call this a *passive* aggregate computation. It is also tried when one of the CHR constraints in the `Goal` of the aggregate is added, reactivated, or removed; this is an *active* aggregate computation. In the banking example of Section 6.1.2, for instance, there is a passive aggregate computation when a new `client` is added, and an active aggregate computation when an `account` is added or removed.

⁴We often use pseudo-Prolog code of the form `Predicate(X_1, \dots, X_n)` to denote the term obtained from adding a series of arguments to a given term `Predicate`. Note that this term may already contain arguments. The first arguments given to the predicates that compute an aggregate can thus be provided by the user. This feature is used in the aggregate expression of Example 6.6 to pass ‘0’ to the unification predicate. This could even be used to pass values from other occurrences in the head to the aggregate computation process.

⁵This is actually a slight simplification of what actually happens. The precise (matching) semantics of the `Result` argument is specified further in Section 6.3.1.5.

<i>Aggregate</i>	<i>Meaning</i>	<i>Aliases</i>
<code>findall(X,G,L)</code>	L is a list of X 's for every match of G	<code>collect</code>
<code>count(G,C)</code>	G matches C times	<code>nb</code>
<code>exists(G)</code>	G exists (<code>count(G,C)</code> , $C > 0$)	
<code>\+G</code>	no G exists (<code>count(G,0)</code>)	<code>none</code>
<code>~G</code>	no other G exists	<code>no</code>
<code>forall(G,C)</code>	for every match of G , condition C holds	<code>implies</code>
<code>min(X,G,M)</code>	M is the minimal X for all matches of G	<code>minimum</code>
<code>min(X,G,M,D)</code>	... with default value D (<code>min/3</code> : no match)	<code>minimum</code>
<code>argmin(X,G)</code>	G is matched such that X is minimal	<code>findmin</code>
<code>takemin(X,G)</code>	like <code>argmin(X,G)</code> , but G is removed	<code>rmin</code>
(<code>max/3</code> , <code>max/4</code> , <code>argmax/2</code> , and <code>takemax/2</code> are analogous)		
<code>sum(X,G,R)</code>	R is the sum of X over all matches of G	
<code>prod(X,G,R)</code>	... product ...	<code>product</code>
<code>avg(X,G,R)</code>	... (arithmetic) average ...	<code>average</code>
<code>stddev(X,G,R)</code>	... standard deviation ...	
<code>var(X,G,R)</code>	... variance ...	<code>variance</code>

Table 6.1: The predefined aggregates offered by our reference implementation. Most aggregates have goal (G ; also C in `forall/2`), element (X), and result (L, M, R, \dots) arguments as defined in the universal aggregate construct.

6.2.2 Common aggregates

Predefined aggregates

Any implementation should offer concise, familiar shorthands for at least the most common aggregates. Table 6.1 lists the aggregates predefined in our reference system. Their declarative meaning should be intuitively clear. Some remarks:

- The `findall/3` aggregate is the CHR counterpart of the well-known Prolog predicate with the same name (see Section 3.1.3 on page 25).
- The `forall/2` construct is not really an aggregate, but rather syntactic sugar for nested aggregates, as explained in Example 6.13 in Section 6.3.1.2.
- The semantical differences between the two negation as absence aggregates are discussed in Sections 6.3.1.8 and 6.3.1.9.
- For all arithmetic aggregates, the X argument must evaluate to a ground arithmetic expression at runtime. Any expression that may be used on the right-hand side of the `is/2` Prolog built-in can be used (cf. Ex. 6.6).
- For most arithmetic aggregates, if no matches for its goal G are found, then the rule is not applicable. Exceptions are `sum/3` and `prod/3`, which evaluate to the respective default values 0 and 1, and `min/4` and `max/4`, where the default value is specified by the user.

Many of these aggregates can be trivially expressed in terms of the universal aggregate construct. In certain cases, further optimisation and non-trivial data structures for optimal incremental maintenance are required. For these and other implementation details, we refer to (Van Weert et al. 2008).

User-defined aggregates

To specify custom aggregates, a user may use the general `aggregate/7` notation directly in the head of rules. However, as this is overly verbose, it is useful to have a macro facility to define abbreviations for commonly used aggregates. The CHR host language usually offers such a facility, e.g. term expansion in Prolog, or C's macro language. We prefer to integrate some macro facility in the CHR language itself for reasons of portability, and to avoid possible conflicts with CHR compilation⁶. The proposed syntax is as follows:

```
:- chr_expansion head ---> body.
```

This replaces any occurrence of `head` in the head of a rule with `body`, where `head` is a single atom, and `body` can be any conjunction of atoms.

Example 6.7. A sum aggregate could be defined as (cf. Example 6.6):

```
:- chr_expansion sum(E,G,R)
   ---> aggregate(=(0),plus,minus,=,E,G,R).
```

Example 6.8. Using this simple macro expansion construct, it is also possible to assign application-specific names to (special cases) of existing aggregates. For instance, in a program that reasons over graphs the in- and out-degree of a node `N` could be defined as:

```
:- chr_expansion in_degree(N,C) ---> count(edge(_,N), C).
:- chr_expansion out_degree(N,C) ---> count(edge(N,_), C).
```

Not only are these user-defined aggregate names more user-friendly than the general `aggregate/7` construct, they also vastly increase the readability and maintainability of the resulting programs.

The above is just a rather primitive, ad-hoc macro facility. More advanced source-to-source transformations are possible using the so-called *meta CHR rules* we introduced for (Van Weert, Sneyers, and Demoen 2008).

6.3 Language Design

In the previous section we have shown how to specify an aggregate, and outlined the basic principles of how aggregate values are computed at runtime. We now

⁶In Prolog, CHR compilation itself is often realised through term expansion.

explore the various language design issues encountered when embedding these aggregates in rule-based programs. This includes issues such as the types of supported aggregate goals, the precise matching semantics of aggregate conditions, and their interaction with the other conditions in the rule's head. In Section 6.3.1, we explore various interesting design alternatives, and motivate our choices. Section 6.3.2 then discusses some significant issues we encountered when adding aggregates to existing CHR systems, and how these are fully resolved in CHR2.

6.3.1 Aggregates for rule-based programs

First some basic properties that have not been stated explicitly:

- If a **Goal** shares variables occurring in the context surrounding the aggregate expression—typically a rule's head, but it could also be the **Result** of another aggregate; cf. Section 6.3.1.2—this gives rise to implicit equality guards. In other words: the same pattern matching semantics is used for these **Goals** as for a normal head.
- A **Result** variable behaves as any other variable in the rule's head (see also Section 6.3.1.5). It can thus be used for matching, in guards, and in the rule's body. Variables introduced in a **Template** or a **Goal**, but not used elsewhere in the rule's head, are in principle local to the aggregate (alternative semantics though are discussed in Section 6.3.1.7), and cannot be used in other left-hand side conditions or the rule's body.

Many of the now following language design issues are (partly) concerned with either the **Template**, the **Goal**, or the **Result** argument of aggregates as well. We denote this in parenthesis in the subsection title.

6.3.1.1 Complex aggregate goals (\rightsquigarrow Goal)

Until now we have only shown examples of aggregates over a simple **Goal**, i.e., consisting of a single CHR constraint. We of course support more complex aggregate goals as well, containing guarded joins of CHR constraints. The same pattern matching semantics applies as for regular CHR heads. An aggregate goal is analogous to any left-hand side in CHR2 (Section 5.1.1), only without any '+' or '-' modifiers. As in CHR2, the aggregate processor can normally distinguish between CHR and built-in constraints in an aggregate goal, and the '?' modifier may be used to clarify or disambiguate as needed.

Example 6.9. The `count((platinum(C), account(_,C,B), B < 0), N)` aggregate counts the number of accounts owned by platinum clients that have negative balances. Its goal is a conjunction of two CHR constraints, joined by a common variable, and a built-in constraint as guard.

6.3.1.2 Nested aggregates (\rightsquigarrow Goal)

Even more expressiveness is realised by allowing *nested aggregates*, that is, aggregate expressions inside the goal of another aggregate. While fully supporting nested aggregates complicates the implementation, the added expressiveness is well worth the effort, as clearly illustrated by the following examples:

Example 6.10. To get the client *C* with the largest total balance, we can use e.g. ‘`argmax(S, (client(C), sum(B,account(_,C,B),S)))`’.

Example 6.11. In constraint solving the *first-fail principle* is a well-known labelling heuristic that, each time a choice-point is created, selects the variable with the smallest domain (Haralick and Elliott 1979). Suppose for instance that a simple Sudoku puzzle solver has a `cand(P,V)` constraint if *V* is a candidate value of the unknown Sudoku cell at position *P*. First-fail labelling could then be implemented using ‘`takemin(N, (cand(P,V), count(cand(P,_),N)))`’. The full Sudoku program is shown in Listing 6.4 of Section 6.5.

Example 6.12. A connected directed graph is Eulerian if for every node, the number of outgoing edges is equal to the number of incoming edges. In Listing 6.2 of Section 6.5, we show this takes many rules to check this property in CHR. Using nested aggregates, this condition is succinctly expressed as:

```
forall(node(N), (count(edge(N,_),X), count(edge(_,N),X)))
```

Or, using the syntactic sugar introduced in Example 6.8:

```
forall(node(N), (in_degree(N,X), out_degree(N,X)))
```

Example 6.13. The `forall` ‘aggregate’ itself is defined as a nested negation:

```
:- chr_expansion forall(G,C) ---> \+(G, \+C).
```

6.3.1.3 Empty heads and goals (\rightsquigarrow Goal)

Rules whose head contain only aggregates, and similarly, aggregate Goals without CHR constraint conjuncts, should be supported as well. This can always be implemented as syntactic sugar using the same technique as before in Section 5.2.1, when normalising CHR² rules—that is, the introduction of a kept occurrence of a special, always-present `$init/0` CHR constraint.

6.3.1.4 Multiple aggregates (\rightsquigarrow Template)

If more than one aggregate occurs in the same rule, the order in which the different aggregates are computed is by default left-to-right. This gives some control on the order in which aggregates are evaluated. Of course, the compiler is allowed to change this order if it finds more optimal join orderings. The join ordering problem is discussed in more detail in Section 8.3.2.7.

We have opted to allow variables occurring in the surrounding context to be used in **Templates**. This does give rise though to nonsensical expressions such as ‘ $\text{min}(X, c(X), Y), \text{max}(Y, c(Y), X)$ ’. This could be interpreted either as being equivalent to either “ $\text{min}(X, c(X), Y), \text{max}(A, (c(A), A == Y), X)$ ”, or “ $\text{max}(Y, c(Y), X), \text{min}(B, (c(B), B == X), Y)$ ”. To avoid confusion, we therefore propose the aggregate processor rejects such expressions.

6.3.1.5 Matching semantics (\rightsquigarrow Result)

In Section 6.2.1, we stated that after the last working value W_n is computed, $\text{Final}(W_n, \text{Result})$ is evaluated. In pseudo-Prolog code, this is equivalent with ‘ $\text{Final}(W_n, F), \text{Result} = F$ ’, that is, the finalised last working value F is simply *unified* with the **Result** argument of the aggregate. To obtain a more natural, uniform semantics, we actually use matching for the **Result** argument of an aggregate instead of unification, completely equivalent to the arguments of regular occurrences (matching or one-way unification is explained in Section 4.1.3).

6.3.1.6 Nondeterministic aggregates (\rightsquigarrow Result)

All aggregates seen so far are (semi-)deterministic, in the sense that given a fixed matching for the remainder of the rule’s head, either a single unique **Result** is computed, or the aggregate fails and has no **Result**. One could imagine though nondeterministic aggregates with multiple correct **Results**.

Example 6.14. For partially ordered **Element** types, extrema aggregates such as **min/3**, **max/3**, **upper/3** and **lower/3** (the latter compute upper and lower bounds) may have multiple equally correct **Results**.

At least three possible semantics could be given to an applicable rule instance containing a nondeterministic aggregate:

1. *Don’t care nondeterminism*: One of the solutions is nondeterministically chosen in a committed choice manner.
2. *Don’t know nondeterminism*: All solutions are tried nondeterministically using search. When for some chosen solution the computation fails, backtracking occurs and a next solution is tried.
3. *Exhaustive application*: A propagation rule instance may fire multiple times, once for each correct **Result**. For other rules, this semantics reduces to committed choice.

The same three semantics can in principle be applied to nondeterministic guards as well. For guards, all current CHR(Prolog) systems we know of have opted for a committed choice semantics. For uniformity (and ease of implementation), we have chosen an analogous semantics for aggregates. But the other two options are definitely worth considering as well (also for CHR guards!).

6.3.1.7 Constructive aggregates (\rightsquigarrow Goal)

As mentioned earlier, the only natural semantics for variables in an aggregate's **Goal** that are shared with the surrounding context is the common matching semantics. For non-nested aggregates, this context is the rule's head, and these variables can thus be used in the rule's guards or body as well. However, it may be interesting to allow other variables, i.e. those that do not occur in the head or the aggregate's **Result**, to be used in a rule's body as well.

Example 6.15. Consider the following simple rule:

```
~c(X) => writeln(X).
```

A possible semantics for this aggregate generates all values X for which no constraint $c(X)$ exists. The problem though is that, in general, there exists an infinite number of X 's that satisfy that condition.

This is related to *constructive negation* in CLP, a challenging issue studied by many researchers, both from a theoretical and implementation point of view (see e.g. Stuckey 1995; Fages 1997).

Example 6.16. Constructive variants of other aggregates are possible as well:

```
min(X,c(X,Y),M) => writeln(Y-M).
```

With a CHR constraint store containing $c(1,2)$, $c(3,2)$, $c(4,3)$, possible constructive answers are 2-1 and 3-4. This is analogous to the `bagof/3` and `setof/3` ISO Prolog predicates (discussed earlier in Section 3.1.3). As with these predicates, explicit existential variables could be introduced (which of course can no longer be used outside the aggregate):

```
min(X,Y^c(X,Y),M) => writeln(M).
```

For the same store, the above rule would simply print 1. In other words, marking all free variables as existential reduces this constructive aggregate to the regular `min` aggregate we have used so far.

Constructive aggregates are a challenging part of future work. Several questions regarding their semantics and implementation are still open.

6.3.1.8 Distinct matches (\rightsquigarrow Goal)

In CHR, different occurrences of the same constraint predicate in a single head are not allowed to match the same CHR constraint. It seems sensible to extend this matching semantics to for instance negation as absence and `exists`.

Example 6.17. This is illustrated by the following family example:

```

    ss @ +parent(X,Y), -parent(X,Y).
    only_child @ +parent(X,Y), ~parent(X,_) => only_child(Y).
    priority ss > only_child.

```

The first rule is a standard set semantics CHR rule for the `parent/2` constraint. The second rule expresses the only child relation. Without CHR’s distinct matches semantics, both rules would need to be rewritten, as the first rule would simply remove all `parent/2` constraints, and the second would never fire.

Similar observations hold for the `exists` aggregate, but less for other aggregates. In our prototype implementation, we only experimented with distinct matching semantics for the ‘`~`’ aggregate. All other aggregates, including ‘`\+`’ and `exists` (for now), allow overlapping constraint matches.

6.3.1.9 Fire-once versus fire-many semantics

This subsection considers the semantics of propagation rules containing aggregates. Traditionally, CHR operational semantics enforce a *fire-once* policy, where every rule instance is allowed to fire at most once. However, when the rule’s head contains aggregates, the question arises whether or not a rule instance should be reapplied if the values of these aggregates change.

Example 6.18. Reconsider for instance this rule from Example 6.4:

```

    new_min @ +c(X), ~(c(Y), Y < X) => min(X).

```

Suppose at some point this rule fires with `X=5`, and that `c(2)` is added next. The rule instance with `X=5` now no longer is applicable. If later `c(2)` is removed again, the intended semantics is clearly that the instance for `X=5` fires again.

We found that for negation as absence, a *fire-many* semantics seems more appropriate (Van Weert et al. 2006a). Negation though is an instance of a specific type of aggregates: if its value changes, the rule’s applicability changes with it. Other aggregates that share this property are e.g. `exists` and `forall`. A suitable fire-many semantics for such aggregates may be stated informally as: “*A rule instance may be applied once each time it becomes applicable*”.

Example 6.19. In retrospect, this semantics may warrant further refinement. Reconsider the `only_child` rules of Example 6.17. When the set semantics rule removes a duplicate `parent/2` constraint, the `only_child` rule, however briefly, has become inapplicable and applicable again. It arguably is not a desirable operational semantics to re-apply in such cases.

In general, the value of an aggregate may change without affecting the rule’s applicability. One attempt to generalise the fire-many semantics is: “*An instance*

may be applied once each time an aggregate’s value changes”. Fire-many semantics may lead to unwanted reapplications, or even non-termination.

In our prototype implementation, aggregates have a fire-once semantics. Only for negation, we have experimented with both options: the ‘\+’ predefined aggregate has fire-once semantics, whereas ‘~’ has fire-many semantics.

6.3.1.10 Aggregates outside rule heads

So far, we only considered aggregate expressions in the head of CHR goals. Clearly, (efficiently) querying the CHR store outside a rule’s head—i.e., in a rule’s body, or even from user code or via interactive queries—is equally useful. In fact, this is frequently requested missing feature of current CHR systems. Most offer only limited support for inspecting the CHR store (iteration over or sometimes counting of all constraints of a given predicate). Efficiently supporting aggregate queries—or other complex queries containing joins etc.—over the CHR constraint database (think SQL) is an important area of future work. The main issue is that such features require considerably different implementation techniques, as the structure of queries is not known in advance. For now, all required queries must be anticipated and encoded explicitly as a CHR rule.

6.3.2 Aggregates in CHR and CHR²

In Van Weert et al. (2006a, 2007), we disclosed many practical issues that surfaced when adding aggregates to the ω_r -based K.U.Leuven CHR system in SWI Prolog. Some are related to the unexpected behaviour of aggregates in ω_r -based programs, others to our implementation based on source-to-source transformation:

1. Due to the sequential, piece-wise execution of rules in ω_r -based systems, rules often fire with aggregate results that reflect inconsistent, intermediate execution states. Common examples are:
 - (a) Modifications, where the old constraint is removed, but the new, modified version is not yet added.
 - (b) The lack of atomic constraint additions often results in unintended runtime behaviour. Sometimes subtle reorderings of constraint conjunctions can fix this, but not always. With the `only_child` rule in Example 6.17, for instance, it is impossible to add both `parent`’s of a child without propagating an `only_child` constraint. Several more examples are given in (Van Weert et al. 2006a, 2007).
2. Examples where the deficient execution control mechanism of traditional systems has proven inadequate to implement aggregates are:
 - (a) The results of nested aggregate must be fully computed before they are used in matchings for an outer aggregate’s goal.

- (b) After a built-in constraint is added, all affected matchings with an aggregate’s goal must be (re)considered before using its result.
- (c) For aggregates with fire-many semantics, the propagation history must be updated before reconsidering rule matchings.

In Van Weert et al. (2006a, 2007), we managed to solve most (not all) of these issues using complex, ad-hoc, low-level encodings. Obviously, this complicated our implementation tremendously. Moreover, in the resulting system, the programmer often has to reason on the low-level ω_r -like execution semantics of aggregates, in order to tune the execution order e.g. by reordering rules or constraint conjunctions, or by adding low-level pragmas.

In CHR2 (and to a lesser extent CHR^{FP}), all these issues can be solved naturally thanks to batch semantics and priorities, regaining all advantages of declarative programming, both in the extended CHR language, as in its (source-to-source) implementation. CHR2’s priority constraints, which allow local priority declarations, are particularly convenient in the context of automatic program transformation because predicting which code still has to be generated often is impractical (e.g. with nested aggregates). Also, in our case, multiple aggregates—with the exception of nested aggregates of course—should be allowed to be computed independent from each other (or even in parallel). CHR^{FP}’s global priority numbers cannot express these constraints.

6.4 Formal Semantics and Properties

6.4.1 Operational semantics

We modify the definition of the transition rules of CHR semantics to deal with (nested) `aggregate/7`-expressions. We use two mutually recursive definitions:

Definition 6.1. With A a conjunction of aggregates, and $S \subseteq \mathbb{S}$, we redefine the set of matching substitutions $\text{matchings}(H, G, A, S, \mathbb{S}, \mathbb{B})$ as follows:

$$\left\{ \theta \mid \text{CHR}(S) = \theta(H) \wedge \mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \exists_{\mathbb{B}}(\theta \wedge G \wedge \forall a \in A : \text{agg_cond}(a, \mathbb{S}, \mathbb{B})) \right\}$$

Definition 6.2. We define the *aggregate condition* as follows, for an aggregate a of form `aggregate`(s, i, f, d, X, G, R), where G consists of conjunctions of CHR constraints C , built-in constraints B , and (nested) aggregates A :

$$\text{agg_cond}(a, \mathbb{S}, \mathbb{B}) = s(V_0) \wedge \bigwedge_{k=1}^n i(V_{k-1}, \theta_k(X), V_k) \wedge f(V_n, R)$$

with V_0, \dots, V_n fresh variables, and $\forall k : \theta_k \in \Theta_k$ with

$$\{\Theta_1, \dots, \Theta_n\} = \{\Theta = \text{matchings}(C, B, A, H, \mathbb{S}, \mathbb{B}) \mid H \subseteq \mathbb{S} \wedge \Theta \neq \emptyset\}$$

The definition is unambiguous if the increment predicate i corresponds to a commutative operator.

The extended definition of `matchings` can, in principle, be plugged in to any ‘theoretical’ operational semantics of CHR. Doing so, however, does not always result in a good fit. Batch semantics, for instance, is a highly desirable property. That is why the ω_p semantics of CHR^{IP} (Section 4.4.2), or the ω_2 semantics of CHR2 (Section 5.2), are better suited to host aggregates than traditional incremental operational semantics such as ω_t (Section 4.2.2). This is discussed in more detail in Section 6.3.2.

Refined operational semantics

Aggregates may also be incorporated into more ‘refined’ operational semantics of CHR, that is, those operational semantics that are based on the concept of active constraints (see Section 4.2.3). Besides the standard ω_r semantics, another example is e.g. ω_{rp} for CHR^{IP} (Section 4.4.2). We specified such refined semantics for CHR with negation as absence in Van Weert et al. (2006a, 2009) In line with our argumentation in Section 5.2, however, we feel that such refined operational semantics should in principle be avoided.

Fire-many semantics

We discussed fire-many semantics for propagation rules with aggregates in Section 6.3.1.9. While it is not immediately clear how to specify such a semantics for aggregates in general, for aggregates like negation as absence, `exists`, and `forall`, it can informally be formulated as: “*A rule instance may be applied once each time it becomes applicable*”. This can readily be accomplished by extending any semantics of CHR with a transition rule that eagerly removes all propagation history tuples as soon as they no longer are applicable (see e.g. Van Weert et al. (2006a, 2007, 2009)).

6.4.2 Logical semantics and formal properties

Both negation as absence and aggregates in general are introduced as operational concepts, much like negation as failure in Prolog (cf. Section 3.1.3). This of course has immediate consequences for the language’s formal semantics and properties. For one, CHR extended with aggregates is clearly no longer monotonic; that is: adding constraints may invalidate previously applicable or applied rule instances. Therefore:

1. the extended language no longer has a direct correspondence to either classical or linear logic (Section 4.2.1), which both are monotonic logics.

2. all results that rely on CHR's monotonicity property are no longer directly applicable. Current confluence checks, in particular, inherently rely on monotonicity, as discussed in Section 4.3.2. Nevertheless, confluence for the extended language remains an interesting and desirable property. In a way, negation as absence actually facilitates writing confluent programs.

Example 6.20. A common pattern in ω_r -based systems looks as follows:

$$\begin{aligned} A, B &\Leftrightarrow C. \\ A &\Leftrightarrow D. \end{aligned}$$

where all capital letters represent conjunctions of CHR constraints. Clearly, such programs are rarely ω_t -confluent. Using negation, an elegant form of confluence can be regained by changing the second rule to the complement of the first: ' $A, \sim B \Leftrightarrow D.$ '

Of course, the same holds for CHR programs that explicitly compute non-monotonic aggregates using on lower-level auxiliary constructs. These programs require some execution control mechanism for correctness.

A detailed study of the logical semantics and other important properties of the extended CHR programs is left as future work.

6.5 Expressiveness case studies

To demonstrate the expressiveness added by aggregates, we now present four different case studies. For four existing programs, we identified the code used to compute (nested) aggregates, and replaced it with the equivalent aggregate conditions. This is shown in Listings 6.1–6.4. The replaced code is indicated using italics and underlining. Note that all programs, including those with aggregates, assume an ω_r -based execution strategy. It is not important, however, to fully understand all programs (see below for references to more detailed explanations); the main point is that the programs with aggregates are more concise, readable, and easier to understand than the original versions.

The four case studies in question are:

DIJKSTRA: a well-known CHR program by (Sneyers et al. 2006a) that implements Dijkstra's single source shortest path algorithm. To obtain the correct runtime complexity, the original program is complemented by a fairly complex CHR-based implementation of a Fibonacci heap (Cormen et al. 2009, Chapter 19). When using aggregates, this is taken care of implicitly by the incremental aggregate implementation (cf. Van Weert et al. 2008).

EULER: a simple program that checks whether a given connected directed graph is Eulerian; we explained this case earlier in Example 6.12.

Listing 6.1 DIJKSTRA: a CHR implementation of Dijkstra’s single-source shortest path algorithm. The original implementation requires an additional implementation of a priority queue with the operations `insert(+,+)`, `decr_key(+,+)`, and `extract_min(-,-)`. The `maintain` pragma indicates the aggregate has to be maintained (the implementation will again use a priority queue; see Van Weert et al. 2008). The `passive` pragma is required by lack of better execution control mechanisms in the ω_r -based prototype.

```

:- chr_constraint edge(+,+,+), dijkstra(+), distance(+,+),
    scan(+), label(+,+), relabel(+,+).
dijkstra(A) <=> label(A,0), scan(A).

scan(A) \ label(A,D) <=> distance(A,D).
scan(A), distance(A,D), edge(A,B,W) ==> relabel(B,D+W).
distance(B,_) \ relabel(B,L) <=> true.
label(B,X) \ relabel(B,L) <=> L >= X | true.
label(B,X), relabel(B,L) <=> label(B,L), decr_key(B,L).
    relabel(B,L) <=> label(B,L), insert(B,L).
scan(A) <=> extract_min(B,_) | scan(B).
scan(_) <=> true.

:- chr_constraint edge(+,+,+), dijkstra(+), distance(+,+),
    scan(+), label(+,+).
dijkstra(A) <=> label(A,0), scan(A).
label(A,X) \ label(A,Y) <=> X <= Y | true.
scan(A) \ label(A,D) <=> distance(A,D).
scan(A), distance(A,D), edge(A,B,W), \ distance(B,_)
    ==> label(B,D+W).

scan(A), argmin(L,label(B,L))#maintain#passive <=> scan(B).
scan(_) <=> true.

```

Listing 6.2 EULER: a CHR program that checks whether or not a connected digraph is Eulerian. The `passive` pragma is added for performance reasons (would not be required if the query had batch semantics).

```

:- chr_constraint node(+), edge(+,+), euler,
    test(+), degree(+,+), get_d(+,?).
euler, node(N) ==> test(N).
euler <=> true.
test(N), edge(N,_) ==> degree(in, 1).
test(N), edge(_,N) ==> degree(out,1).
test(N) <=> get_d(in,X), get_d(out,X).
degree(X,Y), degree(X,Z) <=> degree(X,Y+Z).
get_d(X,Q), degree(X,Y) <=> Q = Y.
get_d(X,Q) <=> Q = 0.

:- chr_constraint node(+), edge(+,+), euler.
euler, forall(node(N),
    (count(edge(N,_)X), count(edge(_,N)X))
    )#passive <=> true.
euler <=> fail.

```

Listing 6.3 HOPCROFT: a CHR implementation of Hopcroft’s algorithm for minimising states in a finite automaton (Hopcroft 1971). The ‘#m’ annotation is a short-hand notation for the ‘#maintain’ pragma, which states that the aggregate’s value has to be maintained incrementally (cf. Van Weert et al. 2008). Understanding the full program is not necessary (see Sneyers (2008, §5.3) for a detailed explanation), simply note that the rules with aggregates are far more readable than their original counterparts (recall from Table 6.1 that `nb/2` is an alias for `count/2`).

```

:- chr_constraint state(+), delta(+,+,+),
   input(+), final(+), init, loop, b(+,+),
   a(+,+,+), l(+,+), k(+), part(+,+), mov(+,+),
   c(+,+), new_l(+,+,+), inv(+,+), fix(+,+),
   add_a(+,+,+), nb_a(+,+,+).

init, final(S) \ state(S) <=> b(1,S).
init \ state(S) <=> b(2,S).
b(I,S), input(A) ==> nb_a(A,I,0), add_a(A,I,S).
delta(_,A,S) \ add_a(A,I,S) <=> a(A,I,S).
add_a(,_,_) <=> true.
a(A,I,_) ==> nb_a(A,I,1).
nb_a(A,I,X), nb_a(A,I,Y) <=> nb_a(A,I,X+Y).
init <=> k(3), add_to_l(1,2), main_loop.
add_to_l(J,K), input(A), nb_a(A,J,X), nb_a(A,K,Y)
  ==> (X <= Y -> l(A,J) ; l(A,K)).
add_to_l(,_) <=> true.
main_loop, l(A,I) <=> part(A,I).
main_loop <=> true.
part(A,I), a(A,I,X), delta(T,A,X) \ b(J,T) <=> bp(J,T).
part(,_) , bp(J,_)#passive \ k(K) <=> do_part(J,K).
do_part(J,K) \ bp(J,T)#passive <=> b(K,T), fix(J,T).
  fix(J,T) \ a(,J,T) <=> nb_a(A,J,-1).
  fix(J,T) <=> true.
do_part(J,K) <=> add_to_l(K,J), k(K+1).
part(,_) <=> main_loop.

:- chr_constraint state(+), delta(+,+,+),
   input(+), final(+), init, loop, b(+,+),
   a(+,+,+), l(+,+), k(+), part(+,+), mov(+,+),
   c(+,+), new_l(+,+,+), inv(+,+), fix(+,+).

init \ state(S), nb(final(S),B) <=> b(2-B,S).

b(I,S), input(A), exists(delta(,A,S)) ==> a(A,I,S).

init <=> k(3), add_to_l(1,2), main_loop.
  add_to_l(J,K), input(A), nb(a(A,J,_)#m, nb(a(A,K,_)#m)
    ==> (X <= Y -> l(A,J) ; l(A,K)).
  add_to_l(,_) <=> true.
main_loop, l(A,I) <=> part(A,I).
main_loop <=> true.
part(A,I), a(A,I,X), delta(T,A,X) \ b(J,T) <=> bp(J,T).
part(,_) , bp(J,_)#passive \ k(K) <=> do_part(J,K).
do_part(J,K) \ bp(J,T)#passive <=> b(K,T), fix(J,T).
  fix(J,T) \ a(,J,T) <=> true.
  fix(J,T) <=> true.
do_part(J,K) <=> add_to_l(K,J), k(K+1).
part(,_) <=> main_loop.

```

Listing 6.4 SUDOKU: a CHR implementation of a simple Sudoku solver. The auxiliary predicate `row_col_box/2` succeeds iff the given coordinates are in the same row, column, or box.

<pre> :- chr_constraint solve, val(+,+), cand(+,+), <u>nb_cand(+,+), solve(+).</u> <u>solve <=> solve(1).</u> solve(<u>N</u>), cand(P,V), <u>nb_cand(P,N)</u> <=> (val(P,V) ; N>1, <u>nb_cand(P,N-1)</u>), solve(<u>1</u>). <u>solve(N) <=> N<9 solve(N+1).</u> solve(<u>_</u>) <=> true. <u>cand(P,_) ==> nb_cand(P,1).</u> <u>nb_cand(P,X), nb_cand(P,Y) <=> nb_cand(P,X+Y).</u> <u>val(P,_) \ nb_cand(P,_) <=> true.</u> val(P,_) \ cand(P,_) <=> true. val(P,V) \ cand(Q,V), <u>nb_cand(Q,N)</u> <=> row_col_box(P,Q) N>1, <u>nb_cand(Q,N-1).</u> </pre>	<pre> :- chr_constraint solve, val(+,+), cand(+,+). solve, <u>takemin(N, (cand(P,V), nb_cand(P,_,N)))</u> <=> (val(P,V) ; N>1), solve. solve <=> true. val(P,_) \ cand(P,_) <=> true. val(P,V) \ cand(Q,V), <u>nb_cand(Q,_,N)</u> <=> row_col_box(P,Q) N>1. </pre>
--	---

Program	original			aggregates			% gained		
	<i>C</i>	<i>R</i>	<i>B</i>	<i>C</i>	<i>R</i>	<i>B</i>	<i>C</i>	<i>R</i>	<i>B</i>
BANKING	6	4	493	5	3	300	17%	25%	39%
DIJKSTRA	6	9	497	5	6	266	17%	33%	46%
+FIBHEAP	+10	+22	+1211	+0	+0	+0	69%	81%	84%
EULER	6	8	351	3	2	141	50%	75%	60%
HOPCROFT	18	19	985	16	14	789	11%	26%	20%
SUDOKU	5	9	456	3	4	245	40%	56%	46%
<i>Average (not counting FIBHEAP)</i>							29%	48%	43%

Table 6.2: Expressivity gained by using aggregates, in terms of the number of CHR constraints (*C*), CHR rules (*R*), and bytes (*B*).

HOPCROFT: a CHR program explained in detail in Sneyers (2008, §5.3) that implements Hopcroft (1971)’s algorithm for minimising states in a finite automaton. The version that uses aggregates is even closer to the standard pseudo-code description given in Sneyers (2008, §5.3).

“Our main point here is that the pseudo-code [...] can be translated quite easily to CHR rules. [...] Although conciseness and readability are obviously somewhat vague notions, we would argue that the CHR program is as concise and readable as the pseudocode description on which it is based. Additionally the CHR program is executable and achieves the optimal $\mathcal{O}(n \log n)$ time complexity. ”

SUDOKU: a straightforward CHR program that solves Sudoku puzzles using the standard first-fail principle, as explained in Example 6.11.

We can quantify the gained conciseness by using aggregates in terms of the number of constraints, rules, and bytes in the program. Table 6.2 lists these numbers. We have also included figures for the motivational BANKING example of Section 6.1.2. In the selected case studies, on average, almost half of the original code dealt with the computation of aggregate values.

In our reference implementation, all aggregate-based programs still retain the correct space and time complexities. This is verified in (Van Weert et al. 2008).

6.6 Related Work

Constructs related to aggregates are found in many languages. In this section we briefly discuss some of them.

SQL SQL is the standard query language for databases, and is well-known to include several aggregate functions (ISO 2003). Because, unlike CHR (cf. Section 4.3.3), SQL is not Turing-complete, these aggregates are very important as they add computational power to the language. The original SQL standard only supports five aggregate functions: `min`, `max`, `count`, `sum`, and `avg`. Practise though showed that users often require to aggregate data in many other ways. To meet this need, all major database systems added numerous other built-in aggregate functions, some of which were standardised in later revisions of the SQL standard⁷. More recently, many database systems also include the possibility to extend the database query language with user-defined aggregates.

⁷E.g. `every` and `any/some` in SQL-99 and several statistical aggregates in SQL-2003.

Rule-based Programming As seen in Chapter 2, production rule (PR) systems have always offered support for negation as absence. Following CLIPS, many support `exists` and `forall` aggregates as well, though not always in a nested fashion. Note that these atypical aggregates do not return a result, but simply represent applicability conditions. Not long after our research was conducted, major PR systems such as Jess and Drools independently started to introduce `accumulate` constructs, analogous to our `aggregate/7`. Of the studied PR systems, Drools is currently the only one to have a full-featured aggregate framework, more or less analogous to our proposal. At the time of writing, Jess 7.1, for instance, does not support:

- predefined or user-definable syntactic shorthands for common aggregates
- efficient incremental aggregate maintenance
- nested aggregates

To implement aggregates, all PR systems extend the Rete network with special node types (cf. Section 2.2.3). Our approach is the first to both incorporate aggregates with lazy rule evaluation, and to employ source-to-source transformations as an implementation technique.

Logic Programming (LP) Semantics of aggregates have been widely studied in the context of LP (Kemp and Stuckey 1991; Pelov 2004). The best-known practical implementations are the *all solutions* predicates `findall/3`, `bagof/3` and `setof/3` of ISO-Prolog (ISO 1995), introduced in Section 3.1.3 (the latter two are nondeterministic and constructive; cf. Sections 6.3.1.6–6.3.1.7). Other aggregates can be implemented in terms of these generic predicates.

More specifically, the semantics and implementation of many types of negation has received tremendous amount of attention in LP and non-monotonic reasoning research. We have already briefly discussed negation as failure in Section 3.1.3, and constructive negation in Section 6.3.1.7. A comprehensive survey of the fundamental approaches of negation in LP is (Apt and Bol 1994).

Functional Programming Aggregates are a special case of *catamorphisms* or *folds* from category theory, which are widely applied in functional programming (Meijer et al. 1991). While aggregates usually consider implicit and unstructured collections of data (sets and multisets), catamorphisms deal with explicit and tree-shaped algebraic data structures. Many laws for fold have been established using various equational reasoning techniques, e.g. for the parallel (or incremental) computation of folds (Hu and Takeichi 1997).

Imperative Programming Imperative (object-oriented) languages often deal with aggregates in a low-level way: by explicit iteration over collections of objects. The elementary *iterator* design pattern captures this concept (Gamma et al.

1995). However, more and more, the need for a higher-level syntax becomes apparent. Mainstream language designers increasingly turn towards declarative (mostly functional) languages for a solution. The C# extension LINQ e.g. offers a SQL-like syntax for querying data structures (Calvert and Kulkarni 2009). Its implementation is based on various higher-order functions, e.g. the generic `Aggregate` function is really a fold. Modern languages such as Erlang, JavaScript, Python, Ruby, and Scala all support fold-like constructs as well⁸.

6.7 Conclusions

In this chapter, we presented aggregates as a powerful new language feature for CHR. As illustrated by a number of case studies, aggregates eliminate the need for tedious, cross-cutting auxiliary code, and increase the language's expressiveness and conciseness considerably. Aggregates integrate seamlessly with CHR2, resulting in a particularly elegant, very high level programming language.

Our aggregate framework concerns a general aggregates infrastructure that not only allows for a rich set of predefined aggregates, but also caters for user-defined application-specific aggregates. We have fully explored the semantical design space for embedding aggregates into a rule-based language, and provided a first, efficient implementation (cf. Van Weert et al. 2008). We also explained how CHR2 resolves all issues earlier reported with our earlier prototypes.

6.7.1 Future work

Many interesting topics of the extended language are still to be researched, ranging from theory to practise. Some were pointed out earlier in this chapter: a deeper investigation of interesting alternative language design choices (constructive or nondeterministic aggregates, fire-many semantics, etc.; cf. Section 6.3), the study of formal logical semantics, as well as important program properties such as confluence and termination (Section 6.4).

Finally, various ways can be investigated to improve the efficiency of our implementation. In Part III, we already fully integrate negation as absence into the optimising compiler. While we still believe a source-to-source approach is a viable technique for the general framework, other common aggregates could be further integrated into the analyses and compilation scheme, exploiting more low-level support from runtime data structures. A crucial topic for future work is also the development of static and dynamic techniques to automatically select the optimal aggregate computation strategy: on-demand or incremental, or maybe even hybrid strategies. This is related to our ongoing research into dynamic constraint store indexing (Section 8.6.1).

⁸A nice overview table of this is found in *Wikipedia, The Free Encyclopedia* at [http://en.wikipedia.org/w/index.php?title=Fold_\(higher-order_function\)&oldid=347910350](http://en.wikipedia.org/w/index.php?title=Fold_(higher-order_function)&oldid=347910350)

Chapter 7

CHR for Imperative Host Languages

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.

— **Donald Knuth** (born 1938)
American computer scientist

CHR is usually embedded in a CLP host language, such as Prolog or HAL (Chapter 3). Real world, industrial software however is mainly written in imperative, object-oriented programming languages. For many problems though, declarative approaches are more effective. Applications such as planning and scheduling often lead to special-purpose constraint solvers. Because a seamless cooperation with existing components is indispensable, these solvers are mostly written in the mainstream language itself. Such ad-hoc constraint solvers are notoriously difficult to maintain, modify and extend.

A multi-paradigmatic integration of CHR and mainstream programming languages therefore offers powerful synergetic advantages to the software developer. A user-friendly and efficient CHR system lightens the design and development effort required for application-tailored constraint systems considerably. Adhering to common CHR semantics further facilitates the reuse of numerous constraint handlers already written in CHR. And conversely, a proper embedding of CHR in a mainstream language enables the use of innumerable existing software libraries and components in CHR programs.

Moreover, imperative host languages are better suited for efficient evaluation of rule-based programs. The indexing data structures and complex nested loops required can be implemented more effectively in an imperative language. This

aspect is explained in detail in Chapter 8. This chapter focusses on language design and integration aspects.

Early CHR embeddings in imperative languages often either lack performance (Abdennadher et al. 2002; Vitorino and Aurelio 2005), or are designed to experiment with specific extensions of CHR (Wolf 2001a). Also, in our opinion, these systems do not provide a sufficiently natural embedding of CHR in the imperative host language. Instead of integrating the specifics of the new host into an elegant combined language, they unduly port the (C)LP host environment as well. This needlessly enlarges the paradigm shift for imperative programmers. We will show that a tighter integration of both worlds leads to a useful and powerful language extension, intuitive to both CHR adepts and imperative programmers.

Overview Section 7.1, reviews the different problems and challenges met when embedding CHR in an imperative host. Next, we outline our views on effectively designing integrated CHR(*imperative*) systems in Section 7.2. This design is followed by the K.U.Leuven JCHR Systems, as demonstrated in Section 7.3. Finally, Section 7.4, compares with related approaches.

7.1 Impedance Mismatch

CHR was originally designed to use a (C)LP language as a host. Integrating it with imperative languages therefore gives rise to particular challenges. Imperative host languages do not provide certain language features used by many CHR programs, such as logical variables, search, and pattern matching (Section 7.1.1). Conversely, the CHR system must be made compatible with the properties of the imperative host. Unlike Prolog for instance, many imperative languages are statically typed, and allow destructive update (Section 7.1.2).

7.1.1 (C)LP language features

Logical variables

Imperative languages do not provide *logical variables* (Section 3.1.1). Unless they have been assigned a value, no reasoning is possible over imperative variables. Many algorithms written in CHR, however, use constraints over unbound logical variables, or require two, possibly unbound variables to be asserted equal (*variable aliasing*; first discussed in Section 3.1.1). The unavailability of a corresponding feature would limit the usefulness of a CHR system in imperative languages. From experience we know that logical variables facilitate several convenient programming patterns, giving CHR an edge over e.g. production rule systems (Chapter 2). Therefore a logical data type, together with the necessary library routines to manipulate it, has to be implemented for the host language.

Built-in constraints

More general than variable aliasing, CLP languages provide *built-in constraint solvers* for CHR. While Prolog provides only one true built-in constraint, equality over Herbrand terms, more powerful CLP systems such as HAL offer multiple types of constraint solvers (see Duck, Stuckey, García de la Banda, and Holzbaur 2003). Imperative languages on the other hand offer no real built-in constraint support. To allow high level programming with constraints in CHR guards and bodies, underlying constraint solvers may need to be implemented from scratch. In Section 7.2.2 we discuss the effective design of built-in constraint solvers that interact with CHR systems.

Symbolic computation

(C)LP languages facilitate the manipulation of symbolic expressions (typically Herbrand terms), which makes them—and consequently CHR(CLP) systems—particularly suited for symbolic computation, parsing, program transformation, and so on. Imperative languages typically have no built-in support for symbolic data (terms). They do, however, offer a much richer set of (compound) data types, which at least for many other applications are more appropriate than symbolic term representations. The challenges that arise when manipulating these with CHR are discussed further in Section 7.1.2.

Pattern matching

CHR uses pattern matching to find applicable rules. In logical languages, pattern matching is readily available through unification¹, even on elements of compound data structures (Herbrand terms). These matches are referred to as *structural matches*. Imperative hosts typically do not provide a suited language construct to perform pattern matching on its (compound) data types.

Search

To solve non-trivial constraint problems constraint simplification and propagation alone is not always enough. Many constraint solvers also require search. As pure CHR does not provide search, many CHR systems therefore implement CHR^\vee (pronounced “CHR-or”), an extension of CHR with disjunctions in rule bodies. In fact, CHR(Prolog) and other CHR(CLP) systems implement CHR^\vee for free, given the built-in support for (at least) chronological backtracking these languages typically offer. This was discussed at length in Section 4.4.4. Providing search for a CHR system embedded in an imperative host, however, requires an explicit implementation of the choice and backtracking facilities.

¹Although CHR’s pattern matching is different from unification, as explained in detail in Section 4.1.3, it is relatively easy to implement matching using the built-in unification facilities of a typical logical language.

We have chosen not to address this issue at this time. In Section 7.4 we show that earlier work extensively studies such combinations of CHR with search (Krämer 2001; Wolf 2005). Their results show the lack of built-in search can be an opportunity, rather than a misfortune, as custom, more powerful search strategies become possible.

There remain however some interesting challenges for future work, as undoing changes made after a choice-point becomes particularly challenging if arbitrary imperative data types and operations are allowed. The only practical solution seems to be a limitation of the host language code used in those CHR handlers that need to support search. We briefly return to this in Section 7.2.1.

7.1.2 Imperative language features

Static Typing

Unlike Prolog, many imperative languages are statically typed. A natural implementation of CHR in a typed host language would also support typed constraint arguments, and perform the necessary type checking. Calling arbitrary external host language code is only possible if the CHR argument types have a close correspondence with those of the host language.

Complex Data Types

The data types provided by imperative languages are typically much more diverse and complex than those used in logical languages. An effective embedding of CHR should support host language data types as constraint arguments as much as possible. In Section 7.1.1 we saw that in (C)LP embeddings, CHR handlers use structural matches on symbolic terms to specify the applicability of rules. Providing structural pattern matching on arbitrary compound data structures provided by imperative languages requires specific syntax, and has certain semantical issues, as discussed in the next three subsections.

Modification

Contrary to logical languages, imperative languages allow side effects and destructive updates. When executing imperative code, arbitrary fields may therefore change. If these are inspected by CHR guards, these modifications may require constraint reactivations. Modifications to a constraint's arguments could also render inconsistent the index data structures used by an efficient CHR implementation (see Section 8.3.2.2). In general it can be very hard or impossible for the CHR handler to know when the content of values has changed. In the production rule literature this is referred to as the *Modified Problem* (da Figueira Filho and Ramalho 2000; Pacht 1995; 2004). We prefer the term *modification problem*, as 'modified problem' wrongfully suggests the problem is modified.

Behavioural Matches

As structural matches over imperative data types are often impractical (see above), guards test for properties of constraint arguments using procedure calls. This is particularly the case for object-oriented host languages: if constraints range over objects, structural matches are impossible if encapsulation hides the objects' internal structure. Guards are then forced to use public inspector methods instead. Matching of objects using such guards has been coined *behavioural matches* (Bouaud and Voyer 2004). So, not only can it be difficult to determine when the structure of values changes (the modification problem), it can be difficult to determine which changes affect which guards.

Non-monotonicity

The traditional specification of CHR and its logical readings assume monotonicity of the built-in constraints, that is: once a constraint is entailed, it remains entailed. When a non-monotonic host language statement is used in a guard, the corresponding rule no longer has a logical reading. This issue is not exclusive to an imperative host language, but certainly more prominent due to the possibility of behavioural matches and destructive updates. A consequence of using imperative data structures as constraint arguments is indeed that, often, they change non-monotonically. CHR rules that were applicable or that have already been applied can thus become inapplicable by executing host language code.

This problem is more a semantical issue than a practical one. It seems inevitable that when using extra-logical host language code, a CHR program has no consistent logical reading. In fact, as seen in Section 3.1.3, this is also true for Prolog. We therefore use the concept of *pure CHR*, which is analogous to that of pure Prolog. Note though that the lack of a logical reading does not diminish the practical usefulness of 'impure', general purpose CHR.

7.2 Integrated CHR(*imperative*) Systems

A CHR system for an imperative host language should aim for an intuitive and familiar look and feel for users of both CHR and the imperative language. This entails a combination of the declarative aspects of CHR—high-level programming in terms of rules and constraints, both built-in and user-defined—with aspects of the host language. As outlined in Section 7.1, such a combination leads to a number of language design challenges. In this section we outline our general view on these issues. Section 7.3 then provides an extensive case study, the K.U.Leuven JCHR system (Van Weert et al. 2005). A second system designed following the same principles, CCHR (Wuille et al. 2007), is briefly discussed in Section 7.4.1.

7.2.1 Design philosophy

Our design philosophy is mostly contrary to the one adopted by related systems, such as HCHR, JaCK and DJCHR. As seen in Section 7.4, these systems limit the data types used in CHR rules to typed logical variables (JaCK) or Herbrand terms (HCHR and DJCHR). Also, they typically severely restrict the interaction with host language code, if at all possible. However, while this limitation to LP data types is overly restrictive and arguably counterintuitive, some limitations may indeed be in order once e.g. search is provided. We come back to this later.

As noted by Frühwirth et al. (1992) in the context of CLP:

“The key aspect in the CLP(X) scheme is to provide the user with more expressiveness and flexibility concerning the primitive objects the language can manipulate. Clearly the user wants to design his application using concepts that are as close as possible to his domain of discourse, e.g. he wants to use sets, boolean expressions, integers, rationals or reals, instead of coding everything as uninterpreted structures, i.e. finite trees [Herbrand terms], as is advocated in logic programming. ”

We believe the key aspect of the CHR(\mathcal{H}) scheme is analogous. The user wants to design his application using concepts familiar to him from the host language. We therefore believe that any natural, practical embedding of CHR should allow CHR rules to contain (arbitrary) statements and expressions from the host language \mathcal{H} . As motivated in the introduction of this chapter, this is indispensable for a seamless cooperation with software components written in the host language. To avoid data type mismatches when calling imperative code, it follows that CHR constraints best range over regular host language data types. Moreover, it avoids the performance penalty incurred by constantly encoding and decoding of data when switching between CHR and host language.

So, whereas in aforementioned related systems all host language data has to be coded as logical variables or terms, we propose the exact opposite. Logical variables and other (C)LP data types, such as finite domain variables or Herbrand terms, can be encoded as host language data types. The CHR compiler should however provide syntactic sugar for (C)LP data types and built-in constraints to retain CHR’s high-level, declarative nature of programming.

Operational semantics

In (Van Weert, Wuille, Schrijvers, and Demoen 2008), we argued for the refined operational semantics ω_r (Section 4.2.3). The left-to-right execution of guards and bodies is familiar to imperative programmers, and eases the interleaving with imperative host language statements. Moreover, to allow an easy porting of existing CHR solvers, support for the same familiar semantics is at least as important as a similar syntax.

From Chapter 5 though, it should be clear that CHR2's ω_2 semantics is a far better choice. Firstly, it possesses all aforementioned advantages of ω_r . Secondly, it combines this with superior priority-based execution control, intuitive even to users unfamiliar with CHR execution or rule-based programming. And thirdly, it allows not only familiar sequential, imperative style programming, but also an optimal integration of logical (batch) conjunctions of constraints. The latter is indispensable for powerful, more advanced language features (Chapter 6).

Opposite design philosophies?

The limitations imposed by systems such as JaCK and DJCHR—discussed at length in Section 7.4—could in principle be motivated by the fact that they need to be able to undo changes made in CHR bodies, either for search or for adaptation. It is in general unclear how to do this effectively for unrestricted imperative code (cf. Section 7.1.1).

Systems such as JaCK and DJCHR thus essentially become very special-purpose libraries, tailored to exactly those applications where CHR(CLP) systems already excel in. Practice shows, however, that even CHR(Prolog) systems are increasingly used for general purpose programming. Arguably, this is even an intrinsic part of their success. And in fact the same can be said about Prolog itself (cf. Section 3.1.3). Such programs do not always use search or adaptation, and can often be expressed just as naturally without term encodings.

In our systems, we have therefore focused on providing a tight, natural integration of imperative host language features with CHR. Our approach leads to elegant integrated languages, that truly play to their combined strengths.

We do believe though that these seemingly opposite philosophies by no means exclude each other. For instance, while we have not yet fully pursued this, we still maintain that our systems can be extended with search. Depending on the implementation techniques used, some restrictions on allowed language constructs may be in order. But to us it is important that these restrictions are only imposed for those rule bodies that need to be ‘backtrackable’.

Both approaches can thus be seen as starting at the two opposite ends of a spectrum from special- to general-purpose CHR systems. The ultimate goal should be to meet somewhere in the middle, combining the best of both worlds.

7.2.2 Arbitrary built-in constraints and solvers

Above, we argued that arbitrary host language expressions should be allowed. In this section, we show that it remains worthwhile to consider constraints separately, and discuss how to model and design built-in constraints and solvers for use in (imperative) CHR systems.

The semantics of CHR assumes an arbitrary underlying constraint system. Imperative languages however offer hardly any built-in constraint support (Sec-

tion 7.1). Typically, only asking whether two data values are equal is supported natively, or testing inequalities of numeric values (or other ordered data types). Solvers for more advanced constraints have to be implemented explicitly.

In any case—whether they either built in the host language itself, or realized as a host language library, or even by another CHR constraint handler (see below)—we call these solvers *built-in constraint solvers*, and their constraints *built-in constraints*. Our imperative CHR systems support a number of predefined built-in constraints, either as abstractions of host-language constructs, or as efficient solver libraries. They moreover have extensible designs, and facilitate the addition of user-defined built-in constraint solvers.

The interaction between a CHR handler and the underlying constraint solvers is well understood, and can be managed effectively. In our design, we follow the approach by Duck, Stuckey, García de la Banda, and Holzbaur (2003). They distinguish three types of interaction:

1. A built-in constraint solver optionally provides procedures for *telling* new constraints. Using these procedures, new constraints can be added to the solver’s constraint store in bodies of CHR rules and the initial query.
2. Similarly, it optionally offers procedures for *asking* whether a constraint is entailed by its current constraint store or not. These procedures are used for constraints that occur in guards.
3. In case constraint can be asked, a built-in solver must finally alert CHR handlers when changes in their constraint store might cause entailment tests to succeed. The CHR handler then checks whether more rules can be fired. Constraint solvers should relieve the user from the responsibility of notifying the CHR handlers, and notify the CHR solver to only reconsider affected constraints. For efficiency reasons, this is typically solved by adding observers² to the constrained variables. This is discussed in more detail in Section 8.3.5.

Example 7.1. Reconsider the LEQ handler from on page 32. The handler uses one built-in constraint, namely equality over logical variables. For an imperative host language, this constraint will not be natively supported, but implemented as a library. All rules use the *ask* version of this built-in constraint to check whether the equality of certain logical variables is entailed. The *antisymmetry* rule is the only rule using the *tell* version of this constraint. Each time two variables are told equal, the CHR handler must be notified that additional matchings may be possible.

Essential in our design is that we do not require all built-in constraints to have both an ask and a tell version. Constraints natively supported by an imperative host language for instance, such as built-in equality and disequality

²The standard observer pattern is explained e.g. by (Gamma et al. 1995).

checks, typically only have an ask version. Also, traditionally, built-in constraints implemented by a CHR handler only have a tell version.

For a CHR constraint to be used in a guard, it requires both an entailment check, and a mechanism to reactivate constraints when the constraint becomes entailed (as explained above). Properly supporting *ask* versions of CHR constraints remains an open research problem. Two initial approaches exist: Schrijvers et al. (2006) researched automatic entailment checking, whereas Fages et al. (2008) more pragmatically propose a programming discipline where the programmer himself is responsible for specifying the entailment checks.

Conclusion We distinguish constraints from arbitrary host language statements because their interaction with CHR handlers is predictable and better understood. They are monotonic, the compiler can reason on which tell versions affect which ask versions, and the modification problem can be solved efficiently and transparently to the user. A secondary reason is that a CHR compiler may support well-established C(L)P-like syntactic sugar to state these constraints (as assumed e.g. in the LEQ handler of Listing 4.1).

7.3 The K.U.Leuven JCHR Systems

The K.U.Leuven JCHR System, JCHR for short, is a state-of-the-art, high-performance CHR(Java) system (Van Weert et al. 2005). Its upcoming successor, JCHR2, is designed to be a next generation CHR2(Java) system. Their architecture follows the general principles outlined in Section 7.2. This section outlines and illustrates some of our more specific language design choices to effectively integrate CHR and Java. For a comprehensive, fully detailed description of JCHR, we refer to the system's user's manual (Van Weert 2006).

The original JCHR system is freely available at Van Weert 2010c, and has already proven useful in several applications (Section 7.3.7). We now first provide a short historical overview of JCHR, and briefly introduce JCHR2.

7.3.1 Historical overview

JCHR was first developed as a master's thesis project (Van Weert 2005). At that time, two other CHR systems for Java existed: JaCK and DJCHR. As discussed in Section 7.4, however, these systems either lacked performance, or implemented a non-standard variant of CHR. The objective was to develop the first state-of-the-art CHR system for a mainstream, imperative host language. From the start, an important goal was to research the advantages such a host could offer with respect to performance, as imperative languages facilitate low-level optimisations and highly optimized data structures, that are hard or impossible to achieve in Prolog.

The resulting K.U.Leuven JCHR System was presented to the CHR community at the 2005 CHR Workshop. From Van Weert et al. (2005):

“User-friendliness is achieved by providing a high-level syntax that feels familiar to both Java programmers and users of other CHR embeddings, and by full compliance to the refined operational semantics. Flexibility is the result of a well thought-out design, allowing e.g. an easy integration of built-in constraint solvers and variable types. Efficiency is achieved through an optimized compilation to Java and the use of a very efficient constraint store.”

JCHR fully leveraged the performance expectations. Van Weert et al. (2005) reported that the system was already up to orders of magnitude faster than other CHR(Java) and state-of-the-art CHR(Prolog) systems.

The JCHR system has been further improved and extended considerably since then. A summary of the system’s version history is given in Table 7.1. From the listed system improvements, four general themes can be ascertained:

1. Natural integration with Java
2. Ease of programming
3. Static program analysis
4. Optimized compilation and evaluation

Throughout this section, we will further highlight the first two themes; the latter two are discussed in later chapters.

A recent development is JCHRIDE, an Eclipse-based integrated development environment for JCHR created by Abdennadher and Fawzy (2008).

The next generation JCHR² system

Over the years, it became increasingly clear that ω_r -based CHR systems have important problems and limitations (Chapters 5–6). The development for JCHR’s successor therefore started early 2009. A primary incentive was a long overdue proper performance comparison with mainstream production rule systems.

The JCHR² effort resulted in the development of the CHR² language discussed earlier in Chapter 5. The current (internal) prototype of JCHR² is a nearly complete implementation of CHR² (nonreactive CHR and static priorities only), extended with negation as absence (Chapter 6). In (Van Weert 2010a), we demonstrated the effectiveness of our design, and showed that JCHR² outperforms prominent production rule systems by up to several orders of magnitude, both in time and in space. We discuss this further in Section 8.4.2.

7.3.2 JCHR handlers

Example 7.2. As an obligatory first example, Listing 7.1 shows JCHR’s version of the LEQ handler. Aside from the additional type and constraint declarations

05/05	1.0.0–1.0.3	<ul style="list-style-type: none"> • Master’s thesis 2004–2005
11/05	1.1.0–1.1.4	<ul style="list-style-type: none"> • Pragma passive
11/05	1.2.0	<ul style="list-style-type: none"> • <code>Constraint</code> + <code>Handler</code> classes
12/05	1.3.0–1.3.3	<ul style="list-style-type: none"> • Anonymous variables (<code>_</code>) in rule heads • <code>Handler</code> implements <code>Collection<Constraint></code>
09/06	1.4.0	<ul style="list-style-type: none"> • Improved command-line tool • Type inference for variables in rule heads • Subsumption analysis • Join ordering • Trace debugger
10/06	1.5.0–1.5.1	<ul style="list-style-type: none"> • Support for packages, static imports, ... • Constraint access modifiers • Never stored analysis • Set semantics analysis • Observation analysis • Delay avoidance
03/08	1.6.0	<ul style="list-style-type: none"> • On-the-fly compilation • Singleton constraint optimisations • Functional dependency analysis • Improved built-in equality solvers
?	1.7.0	<ul style="list-style-type: none"> • Refactoring to <code>be.kuleuven.jchr</code> packages • Tell-by-asking • Specialized indexes • Conditional indexes • Dynamic never stored optimisation • Lazy indexing • Optimal reactivation prevention (Van Weert 2008b)

Table 7.1: Version history of the K.U.Leuven JCHR System, with a selected overview of advancements. A more comprehensive list, as well as more information on the above improvements, can be found at (Van Weert 2010c).

Listing 7.1 A JCHR implementation of the LEQ handler.

```

package be.kuleuven.jchr.examples.leq;
import be.kuleuven.jchr.runtime.*;

public handler leq<T> {
    public solver EqualitySolver<T> builtin;
    public constraint leq(Logical<T>, Logical<T>) infix =<;
    rules {
        reflexivity @ X =< X <=> true;
        antisymmetry @ X =< Y, Y =< X <=> X = Y;
        idempotence @ X =< Y \ X =< Y <=> true;
        transitivity @ X =< Y, Y =< Z ==> X =< Z;
    }
}

```

required, the original CHR(Prolog) rules can be used almost verbatim. It uses several of JCHR's distinguishing features which we discuss in this section.

JCHR handlers are completely analogous to Java class, interface and enum type definitions. As Listings 7.1–7.3 show, the actual handler definition has the exact same block structure as any Java type definition. That is: a **handler** declaration is embedded into Java at exactly the same level as for instance **class**, **interface**, or **enum** blocks. A `.jchr` source file thus also starts with optional package and import statements (static imports are also supported), all completely equivalent in semantics to their Java counterparts. JCHR² further strengthens the analogy with Java classes even further, and conveniently allows handlers to declare user-defined Java fields and methods.

A JCHR handler moreover declares a number of JCHR constraints, possibly which built-in constraint solvers to use, and of course a number of JCHR rules. These CHR-specific declarations are discussed further in Sections 7.3.3–7.3.4.

Generic handlers

JCHR is a statically typed CHR dialect. The system fully supports Java's generic types though (Bracha 2004; Gosling, Joy, Steele, and Bracha 2005). This eases e.g. the transition from untyped Prolog to strongly typed Java. To the best of our knowledge the K.U.Leuven JCHR system is the first typed CHR-system that adequately deals with polymorphic handlers this way.

Example 7.3. The LEQ handler, for instance, defines inequality constraints over any Java reference type `T`.

Listing 7.2 A JCHR2 encoding of the MERGESORT handler.

```

public handler mergesort<T extends Comparable<? super T>> {
    local constraint arrow(+T From, +T To) infix '->';
    public constraint merge(int, +T);

    X '->' A \ X '->' B <=> A < B | A '->' B;
    merge(N,A), merge(N,B) <=> A < B | merge(N+1, A) & A '->' B;
}

```

Example 7.4. Generics are relatively powerful. Listing 7.2 shows a JCHR2 handler implementing mergesort over any Java type `T` implementing the standard Java `Comparable` interface. The language constructs used to bound the type parameter `T` are analogous to regular Java generics.

Note also JCHR2's streamlined syntax: JCHR's original `rules` block has been deprecated, and arithmetic is fully supported by the front-end.³

7.3.3 JCHR constraints

In JCHR, all constraints—both built-in and user-defined—range over plain Java types (cf. Section 7.2). All Java types are allowed in principle, from primitive and enum types, to regular reference types (i.e. interface and classes). For all constraints the familiar CLP prefix notation is used, and for binary constraints infix notation is supported as well (Listings 7.1–7.2). As in most CHR systems, all JCHR constraints have to be declared explicitly. Unlike CHR(Prolog) systems though, statically declaring argument types for constraints is mandatory.

Example 7.5. Listing 7.3 lists the JCHR2 version of the classic GCD CHR handler, which computes the greatest common divider of any number of (positive) integer values. The declared `gcd/2` constraint has a single argument of Java's primitive `long` type. If, for instance, larger numbers would be required, this could simply be replaced by e.g. the built-in class `java.math.BigInteger`. The '`M >= N`' guard would remain valid, as JCHR knows to test this built-in constraint as '`M.compareTo(N) >= 0`' (see Section 7.3.5). In this case, '`M-N`' still would have to be changed to '`M.subtract(N)`' though.

Example 7.6. In line with our motivation in Section 7.2.1, logical variables are simply implemented as Java classes— see e.g. the LEQ handler in Listing 7.1.

³Compare also with Prolog's version: '`N1 is N+1, merge(N1,A)`'!

Listing 7.3 A JCHR² implementation of the GCD handler.

```
public handler gcd {
    public constraint gcd(long n);
    -gcd(0);
    -gcd(M), +gcd(N), M >= N => gcd(M-N);
}
```

Encapsulation

The standard Java encapsulation mechanism is used to restrict access to a handler's constraint store. JCHR therefore supports Java-like access modifiers for JCHR constraints. Basically, two different operations are restricted: adding new constraints to the store, and inspecting the constraint store. The semantics of the standard Java access modifiers is obvious. Next to these, JCHR also supports the `local` modifier, which means that anyone can inspect the store of that constraint, but the constraints can only be told locally by the handler itself.

Example 7.7. The DIJKSTRA handler adapted from (Sneyers et al. 2006a) contains idiomatic use of the three basic access modifiers:

```
public constraint edge(int,int,int), dijkstra(int);
local constraint distance(int,int);
private constraint scan(int,int), relabel(int,int);
```

`dijkstra/1` and `edge/3` are declared **public**, as they constitute the user's *input* graph. The `distance/2` constraints are **local**, as they are the computed *output* of the program. Both remaining constraints are internal auxiliary constraints, and are completely hidden from external code.

The potential benefits of encapsulation for the programmer are well-known from software engineering. The JCHR compiler also uses them to infer some basic control flow information. To the best of our knowledge, we are the first to make this important distinction between public and private constraints.

7.3.4 An integrated CHR(Java) system

Next to constraints, JCHR guards and bodies may contain most Java statements and expressions.⁴ Naturally, in guards, next to built-in ask constraints only those expressions can be used that return `boolean` values.⁵

⁴JCHR originally only supported a very limited subset of expressions and statements due to restrictions in the parser's design. In JCHR² these restrictions have mostly been eliminated. JCHR² moreover always allows arbitrary Java code in the form of `{...}` blocks.

⁵More precisely: types that can be coerced to `boolean`—for instance `Boolean`, `LogicalBoolean` and `Logical<Boolean>` are also allowed.

The modification problem In Section 7.1.2, we introduced the modification problem. For built-in constraints this is solved as outlined in Section 7.2.2, that is, using the standard observer design pattern (Gamma et al. 1995). Constraining types such as `Logical` implement an interface called `ConstraintObservable`, through which interested JCHR constraints register. The built-in solvers then notify these constraints of relevant changes.

Any user-defined class can similarly implement this interface. The user is responsible for ensuring that the relevant notification events are sent to the observing JCHR constraints. JCHR2 also support other alternatives. Firstly, JCHR2 supports more methods for selectively reactivating subsets of JCHR constraints explicitly from user-code. And secondly, it supports so-called bound properties of JavaBeansTM, a standardized, more general observer-based modification notification mechanism for Java (Hamilton et al. 1997).

However, in many cases, the outcome of guards cannot or does not change. JCHR therefore uses the notion of *fixed* methods and types. Basically, fixed methods consistently return the same value, and for fixed types all public inspector methods are fixed. Many frequently used Java types are fixed: `String`, the eight ‘wrapper’ types (`Integer`, `Boolean`, ...), `BigInteger` and `BigDecimal`, `URL`, etc. Any method and type can be declared fixed, for instance by using the `@JCHR_Fixed` annotation.

Often also when non-fixed types are used, the user nevertheless knows the relevant objects are never changed (at least not in such a way that it affects rule guards). For these cases, JCHR offers the ‘+’ type modifier. In the MERGE handler of Listing 7.2, for instance, such modifiers indicate (promise) that the ‘=<’ order used in the guards does not change over time. Another typical example is the `java.util.Date` class. While it is mutable, in many typical uses the date value nevertheless never changes once fixed.

7.3.5 Built-in constraints and solvers

As indicated earlier, JCHR’s built-in constraints range over plain Java types. Built-in constraints and solvers are modelled as proposed in Section 7.2.2.

Built-in Java ask constraints

By default, JCHR offers the so-called Java built-in constraints shown in Table 7.2. These correspond directly to typical ‘constraints’ found in Java. For several of these, the JCHR constraints provide convenient, more high-level syntactic sugar: e.g. ‘`A.equals(B)`’ becomes ‘`A = B`’, and ‘`A.compareTo(B) < 0`’ simply ‘`A < B`’ (cf. Example 7.5 and Listing 7.2). Only ask versions of these constraints are supported. To allow tell-like versions of these and other ask-only constraints though, we recently added support for a form of negation as failure to JCHR.

Primitive types ^a		Reference types ^b		Comparables ^c	
<i>Prefix</i>	<i>Infix</i>	<i>Prefix</i>	<i>Infix</i>	<i>Prefix</i>	<i>Infix</i>
eq	= or ==	eq	= or ==		
neq	!=	neq	!=		
		ref_eq	===		
		ref_neq	!==		
geq	>=			geq	>=
gt	>			gt	>
leq	<= or =<			leq	<= or =<
lt	<			lt	<

Table 7.2: The Java built-in ask constraints supported by JCHR. They can all be written both prefix and infix.

^aFor `boolean` primitives, only equality and inequality can be asked.

^bBy default, object equality (`eq` and `neq`) is tested using the `equals` method; for `enum` types reference comparison (`==`) is used. Reference comparison can be enforced via the `ref_eq` (`===`) and `ref_neq` (`!==`) constraints. As in Java, reference (dis)equality can only be tested if the static types of the operands are comparable (possibly after coercion).

^cThese are all types assignable (after coercion) to `java.lang.Comparable<T>`. Commonly used Comparables include `String`, `enum` types, `Boolean`, `BigInteger`, `Date`, etc. For equality constraints, JCHR treats them as a reference type.

Example 7.8. This notation for instance allows more convenient variants (to the right) of the following very common JCHR patterns (to the left):

$$\begin{array}{ll}
 H_1 \implies !g_1 \mid \text{fail}. & H_1 \implies g_1. \\
 H_2 \iff !g_2 \mid \text{fail}. & \rightsquigarrow H_2 \iff g_2. \\
 H_2 \iff \text{true}. &
 \end{array}$$

In this JCHR pseudo-code, ‘!’ denotes negation (as in Java), and g_1 and g_2 are ask-only constraints. The new ‘tell-by-asking’ option allows g_1 and g_2 to be used in the body nevertheless. The semantics is equivalent to the original program: if the built-in constraints do not hold, the computation fails.

User-defined solvers

JCHR is designed to be extensible with user-defined built-in solvers. These may override the above default constraints, or implement entirely new built-in constraints. The following example illustrates the basic principle:

Example 7.9. The K.U.Leuven JCHR System contains an efficient reference implementation for equality over logical variables. Its interface declaration is shown in Listing 7.4. It declares a single binary `eq` constraint which can also be written using infix notation. This solver is used in the `LEQ` example of Listing 7.1.

Listing 7.4 The declaration of a generic built-in equality constraint solver interface using annotations.

```

@JCHR_Constraint(
    identifier = "eq", arity = 2, idempotent = YES,
    ask_infix = {"=", "=="}, tell_infix = "=",
    negatedIdentifier = "neq", negatedInfix = "!="
)
public interface EqualitySolver<T> {
    @JCHR_Tells("eq") public void tellEqual(Logical<T> X, T val);
    @JCHR_Tells("eq") public void tellEqual(T val, Logical<T> X);
    @JCHR_Tells("eq") public void tellEqual(Logical<T> X, Logical<T> Y);
    @JCHR_Ask("eq") public void askEqual(Logical<T> X, T val);
    @JCHR_Ask("eq") public void askEqual(T val, Logical<T> X);
    @JCHR_Ask("eq") public void askEqual(Logical<T> X, Logical<T> Y);
}

```

The `solver` declaration tells the JCHR compiler to use an `EqualitySolver<T>` as a built-in solver. From the annotation, the JCHR compiler knows to use the `askEqual` method to check the implicit equality guards, and to use the `tellEqual` method in the body of the *antisymmetry* rule.

Using these `@JCHR_Constraint` annotations, the JCHR compiler can be informed of several other important properties, most only applicable to binary constraints: transitivity, (anti/a)symmetry, (ir/co)reflexivity, etc.⁶ Such knowledge is required for a variety of optimisations, most notably proper guard reasoning (Section 8.3.6.2). JCHR is the first CHR system that can be informed of such useful meta-data for arbitrary constraints this way.

7.3.6 Using a JCHR handler

Using a JCHR handler from Java is straightforward. For each handler `foo`, the JCHR compiler generates a corresponding handler `FooHandler`, with for each of the (non-private) JCHR constraints an inner class. We use the following example to illustrate how easy it is to create JCHR handlers, and to use them in tandem with built-in constraint solvers.

Example 7.10. Listing 7.5 contains an example usage of the classes generated for the `LEQ` handler of Listing 7.1 and the built-in solver of Listing 7.4.

First a new built-in `EqualitySolver` and an `LeqHandler` are created. Note that the constructor of the `LeqHandler` class requires a built-in constraint solver

⁶For the example in Listing 7.4, the default values derived by the system are correct.

Listing 7.5 A code snippet illustrating how the JCHR LEQ handler and equality built-in solvers are called from Java code.

```

1  ...
2  EqualitySolver<Integer> builtin = new EqualitySolverImpl<Integer>();
3  LeqHandler<Integer> handler = new LeqHandler<Integer>(builtin);
4  Logical<Integer> A = new Logical<Integer>(),
5      B = new Logical<Integer>(), C = new Logical<Integer>();
6  handler.tellLeq(A, B); // A =< B
7  handler.tellLeq(B, C); // B =< C
8  handler.tellLeq(C, A); // C =< A
9  // all CHR constraints are simplified to built-in equalities:
10 assert handler.getLeqConstraints().isEmpty();
11 assert builtin.askEqual(A, B);
12 assert builtin.askEqual(B, C);
13 assert builtin.askEqual(A, C);
14 ...

```

to be provided (line 2). In this particular example, we will be solving constraints over `Integer` objects.

Adding JCHR constraints to the handler is just as straightforward (lines 6–8), be it inadvertently more verbose than from within JCHR rules. Logical variable objects also have to be created first (lines 4–5). Constraint objects can only be created by JCHR handlers; constraints are added by calling the generated `tell` methods (if encapsulation permits).

In lines 10–13 it is verified that all JCHR constraints are indeed simplified to built-in constraints. A number of inspector methods are generated (provided encapsulation permits), that allow the final JCHR constraint store to be inspected. These methods return objects implementing the standard `java.util.Collection` interface. In the example, the `getLeqConstraints()` method returns an instance of `Collection<LeqHandler.LeqConstraint>`. In fact, the handler object itself implements `Collection<Constraint>`, with `Constraint` an interface implemented by all generated JCHR constraint classes. Hence, in our example, `handler.isEmpty()` would also have been correct (line 10).

Constraint systems Unlike CHR(Prolog) systems, JCHR allows multiple constraint systems of cooperating solvers and handlers to co-exist independently. Each handler belongs to a *constraint system*. If not assigned a constraint system explicitly; a default constraint system object is used.⁷

⁷By default, one constraint system is created (on demand) per execution thread.

Debugging JCHR programs The JCHR and JCHR2 systems both come with experimental though already useful debugging tools. JCHR handlers can be configured to generate trace events at rule applications, constraint additions, activations and removals, and so on. These events can be logged, used to gather runtime statistics, or for more advanced debugging purposes. The JCHR system for instance contains a debugger with a Swing-based GUI that visualizes the JCHR constraint store, supports coarse-grained breakpoints and event-per-event execution of JCHR programs. JCHR2 at the moment only has a command-line debugger, but allows more fine-grained breakpoints, as well as regular-expression based querying of the constraint store at each step.

7.3.7 Applications

Over the years, JCHR has been used by several people, both from academia and from industry.

Of course, JCHR is first and foremost developed as a research vehicle, to demonstrate and further improve the language and architecture proposed in this dissertation. JCHR has been used extensively for benchmarking CHR's performance (e.g. by Sneyers et al. 2006a; 2009; and Christiansen, Westh, Basbous, and Christiansen 2006), and for researching new compiler optimisations (Van Weert et al. 2008; Van Weert 2008b).

JCHR has also been applied for teaching purposes in declarative programming language courses and student projects.

Although still mostly a proof-of-concept research prototype, JCHR has been proven robust enough for practical applications. It is used in at least two commercial applications, both in the area of software verification. BSSE System and Software Engineering⁸, a German company specialising in the discipline of full automation of software development, once used JCHR for the generation of test data for unit tests. And Agitar Technologies⁹, a US-based company with worldwide distributors and resellers, uses JCHR in their flagship product family AgitarOne. As reported by Daniel and Boshernitsan (2008):

“ AgitarOne is a comprehensive unit testing product for Java developed at Agitar Software. It combines tools for software agitation (a technique for exploratory testing), regression test generation, reporting, and continuous integration. Test generation is supported in AgitarOne by two distinct test generation engines [Agitar's Agitator and Agitar's Mockitator]. [...] Agitar's Mockitator (internal code-name) is a static test generation engine that creates regression tests based on static program analysis. [...] The Mockitator engine generates test inputs statically by constructing a chain

⁸<http://www.bsse.biz/>

⁹<http://www.agitar.com/>

of path conditions for each code path that needs to be covered. As a code path is traversed backward, the conditions are translated into a system of constraints to be incrementally processed by a propagating constraint solver based on constraint handling rules. This arrangement, previously proposed in testing literature, enables early rejection of infeasible paths.”

7.4 Related work

Even though (C)LP remains the most common CHR host language paradigm, an increasing number of other CHR implementations have appeared. We introduced these earlier in Section 4.5; here we further discuss related CHR embeddings in imperative and functional host languages, focussing on how they deal with the issues raised in Section 7.1. We pay extra attention to the CCHR system by Wuille et al. (2007): being heavily inspired by JCHR, it most closely resembles our work.

Of course, countless other declarative paradigms have been integrated and compiled to imperative host languages. We only consider *production rules*, introduced in Chapter 2, because this formalism is most closely related to CHR.

7.4.1 CHR in C

CCHR (Wuille, Schrijvers, and Demoen 2007) is an integration of CHR with the programming language C, strongly inspired by JCHR both in design and implementation strategy (Wuille 2007). It thus follows several of the principles outlined in Section 7.2. CHR code is embedded into C code by means of a `cchr` block. This block can not only contain CCHR constraint declarations and rule definitions, but also additional data-type definitions and imports of host language symbols. Host language integration is achieved by allowing arbitrary C expressions as guards, and by allowing arbitrary C statements in bodies. Functions to add or reactivate CHR constraints are made available to the host language environment, so they can be called from within C.

Constraint arguments are typed, and can be of any C data type except arrays. Support for logical data types is provided, both in the host language and within CHR blocks. CCHR does not have a concept of built-in constraint solvers as introduced in Section 7.2.2. All ‘ask’ requests are simply host-language expressions, and ‘tell’ constraints are host-language statements, which have to be surrounded by curly brackets. It is however possible to declare macro’s providing shorter notations for certain operations, a workaround for C’s lack of polymorphism. When a data type is declared as logical, such macro’s are generated automatically.

Rules follow the normal CHR(Prolog) syntax, yet are delimited by a semicolon instead of a dot. This latter would cause ambiguities since the dot is a C operator.

Listing 7.6 A CCHR implementation of the LEQ handler.

```

cchr {
  logical log_int_t int;
  constraint leq(log_int_t,log_int_t);

  reflexivity @ leq(X,X) <=> true;
  antisymmetry @ leq(X,Y), leq(Y,X) <=> { telleq(X,Y); };
  idempotence @ leq(X,Y) \ leq(X,Y) <=> true;
  transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z);
}

void test(void) {
  cchr_runtime_init();
  log_int_t a=log_int_t_create(), b=log_int_t_create(),
           c=log_int_t_create();
  leq(a,b); leq(b,c); leq(c,a);
  int nLeqs=0; cchr_consloop(j,leq_2,{ nLeqs++; });
  assert(nLeqs==0);
  assert(log_int_t_testeq(a,b));
  assert(log_int_t_testeq(b,c));
  assert(log_int_t_testeq(c,a));
  log_int_t_destruct(a);
  log_int_t_destruct(b);
  log_int_t_destruct(c);
  cchr_runtime_free();
}

```

CCHR implements the standard refined operational semantics.

Example 7.11. Listing 7.6 shows how to implement the LEQ handler in CCHR. It is equivalent to the JCHR/Java code shown in Listings 7.1–7.5. The first line starts the `cchr` block. The next line declares `log_int_t` as a logical version of the built-in C data type `int`. The third line declares a `leq` constraint that takes two logical integers as argument. The four rules of the LEQ handler look very similar to those of Listing 4.1. Equality of logical variables is told using the generated `telleq()` macro.

The `test()` function shows how to interact with the CHR handler from within C. The first line of the function initializes the CHR runtime. The next line creates three `log_int_t` variables (`a`, `b` and `c`), and is followed by a line that adds the three `leq` constraints `leq(a,b)`, `leq(b,c)` and `leq(c,a)`. The next line counts the number of `leq` constraints left in the store. The next four lines assert that no CHR constraints are left, and that all logical variables are equal (in C, if the argument of `assert` evaluates to 0, the program is aborted and a diagnostic

error message is printed). The function ends with the destruction of the logical variables used, and the release of all memory structures created by the CCHR runtime.

Conclusion CCHR truly is much like its host language: a quite low-level (for CHR standards that is), but potentially extremely efficient CHR system. It follows the general design philosophy motivated in Section 7.2.1, though not (yet) the design principles of Section 7.2.2.

7.4.2 CHR in Java

Aside from the K.U.Leuven JCHR systems, there exist at least three other CHR(Java) systems. These are introduced earlier in Section 4.5.3. In this section, we provide some additional comments on their language design.

In JaCK (the Java Constraint Kit; Abdennadher et al. 2001, 2002), CHR constraints only range over typed logical variables. All Java objects thus have to be wrapped in logical variables. Only static Java methods can be called from a rule's body. The CHORD system took a similar approach.

In DJCHR (Dynamic JCHR; Wolf 2001a), constraints range only over Herbrand terms. Integration of the host language in the CHR rules is not supported. The system seems mainly created to experiment with the incremental adaptation algorithm of (Wolf, Gruenhagen, and Geske 2000).

Both JaCK and DJCHR were extended to support a wide range of search strategies (Krämer 2001; Wolf 2005). As discussed earlier in Section 4.5.3, in both systems search has to be implemented in Java, orthogonally to the actual CHR handlers. Interestingly, (Wolf 2005) clearly shows that the use of advanced search strategies can be more efficient than a low-level, host language implementation of chronological backtracking (as in Prolog).

7.4.3 CHR in functional languages

When embedding CHR in functional languages, many of the same challenges are met. Typically, functional languages are statically typed, and do not provide search or built-in constraints. Structural matching on compound host language data on the other hand is mostly readily available (nevertheless, none of the CHR(Haskell) systems seem to fully exploit this).

HCHR provides a type-safe embedding of CHR in Haskell, leveraging the built-in Haskell type checker to type check HCHR programs (Chin, Sulzmann, and Wang 2003). HCHR constraints only range over typed logical variables and terms, encoded as a polymorphic Haskell data type. Unification and matching functions are generated automatically for each type (this is similar to the approach taken by CCHR, cf. Section 7.4.1). Haskell data structures therefore have to

be encoded as terms when used in a HCHR constraint, and reconstructed again when retrieving answers. No Haskell functions can be called from HCHR rule bodies, which seems inherent to their approach to type-safety:

“One of [the enhancements of HCHR we are currently looking into] is to be able to call Haskell functions from CHR rules. This has been shown to be important in many applications. However, this is non-trivial. If function calls instead of terms are allowed in the rules, we will not be able to build the unifiers because unification is defined only on terms.”

The Chameleon system (Stuckey, Sulzmann, and Wazny 2004) is a Haskell-style language that incorporates CHR. It has been applied successfully to experiment with advanced type system extensions (Stuckey and Sulzmann 2005). Chameleon’s back-end CHR solver is HaskellCHR (Duck 2004). To allow Prolog-style Herbrand terms with logical variables, this system includes a WAM implementation for Haskell, written in C. It is the only Haskell CHR system we know of to provide search (chronological backtracking to be precise). HaskellCHR is not intended to be used stand-alone, but as a back-end to Chameleon.

The more recent TaiChi (Boespflug 2007) system, as well as both STM-based parallel CHR(Haskell) prototypes, Concurrent CHR (Sulzmann and Lam 2008) and STMCHR (Stahl and Melnikov 2007), all lack syntactic preprocessing. They are basically interpreters, for which the user has to explicitly encode CHR rules using Haskell data types. In this encoding, the body is typically limited to a Haskell list of CHR and built-in constraints of some fixed type, thus prohibiting any interaction with the host language.

Conclusion Even though both HCHR and Chameleon provide syntactic preprocessing, most CHR(Haskell) typically simply encode CHR(LP) rules as Haskell data types. They moreover do not allow interaction with Haskell, or support built-in constraints beyond term equality. In our opinion, none of the CHR(Haskell) approaches we considered have therefore succeeded at obtaining a useful general-purpose language extension.¹⁰

7.4.4 Production Rule Systems

Modern Java-based production rule systems such as Jess (Friedman-Hill 2003, 2010) and Drools (Browne 2009; Bali 2009; JBoss 2010) seamlessly integrate with the host Java. In this respect, their language design philosophy is similar to ours—though naturally without the concepts of logical variables or built-in constraints. Admittedly, these systems, and Drools in particular, offer much more

¹⁰As discussed in Section 4.5, Chameleon of course is very useful (Stuckey and Sulzmann 2005), but it is created for a very specific purpose only (the design of type system extensions).

powerful pattern matching syntax for Java objects. Drools for instance supports the full MVEL expression language (Brock et al. 2010) to specify matchings and guard conditions.

Interestingly, in Jess and Drools, Java objects can be asserted directly into the working memory, and rules then match on these objects directly. In CHR speak: objects take up the role of CHR constraints. In Drools this is even the default mode of operation.

Example 7.12. Systems such as Drools excel at matching Java objects in rule conditions. Future CHR(Java) systems should take inspiration from this. As a simple example, for objects of class `Person`, a rule condition of form `'Person(name = "Peter")'`—note the capitalisation—could for instance be syntactic sugar for `'$person(P), ?(P.getName() == "Peter")'`. Extend JCHR2 with similar and other object matching syntax is part of future work.

To (partially) solve the modification problem, a technique called *shadow facts* is commonly used.¹¹ Consistency of the Rete network (cf. Section 2.2.3) is maintained by keeping a clone of any modifiable object referred to by facts in an internal data structure. The user remains responsible for notifying the rule engine of any changes to these objects made outside rule program. The techniques used are similar to those we are developing for JCHR2:¹² either modifications are signalled via fact handles obtained when the data was first asserted, or JavaBeansTM bound properties are used (Hamilton et al. 1997).

¹¹Drools 5 seems to have abandoned this technique, replacing it with more efficient modification handling.

¹²At the time of writing, JCHR2's support is still in a very experimental stage. No doubt more can still be learned from studying the approach taken by production rule engines.

Part III

Optimising Implementation of CHR

Chapter 8

Optimising Compilation and Lazy Evaluation

Efficiency is intelligent laziness.

— **Arnold Glasgow** (1905-1998)
American humorist and author

Considerable research has been devoted to the efficient compilation and execution of CHR programs, mostly with Prolog as the host language. An early, very influential implementation was the SICStus implementation by Holzbaaur and Frühwirth (2000a). Its operational semantics was the basis for the refined operational semantics ω_r (see Section 4.2.3), and its compilation scheme has been adopted by state-of-the-art systems such as HALCHR (Holzbaaur et al. 2005; Duck 2005) and K.U.Leuven CHR (Schrijvers and Demoen 2004b; Schrijvers 2005).

In recent years, CHR research has produced a large body of novel static program analyses and optimisation techniques to further improve the performance of such ω_r -based CHR systems: Duck 2005; Duck and Schrijvers 2005; Holzbaaur et al. 2005; Schrijvers 2005; Schrijvers et al. 2005; Sneyers et al. 2006b; 2008; De Koninck and Sneyers 2007; Van Weert 2008b; Sarna-Starosta and Schrijvers 2008a; 2008b. Almost all these contributions were developed and described for the aforementioned CHR(HAL) and CHR(Prolog) systems.

Other recent developments include the introduction of improved control mechanisms and operational semantics (Chapter 5) and expressive new language features (Chapter 6). The impact of these augmentations on CHR's compilation and optimisation techniques had only been scarcely studied. Sole exception is the compiler for CHR^{TP} developed by De Koninck (2008), which introduces and incorporates several important optimisations.

The contributions we make in this chapter can be summarised as follows:

1. We show how to port the classic CHR(Prolog) compilation scheme to an imperative setting. CHR evaluation revolves around nested loops and efficient index data structures. We explain how to fully exploit the added technical advantages imperative target languages therefore offer. Later, in Chapter 9, we identify and address one of their main technical disadvantages (related to the performance of recursion in the generated code).
2. We provide a clear survey that places the many optimisations developed, described in as many different publications, in one coherent framework. Where appropriate we explain interactions and trade-offs between them. Even though introduced and illustrated for an imperative host language, the accessible overview provided in this section is useful for any reader interested in optimised compilation of rule-based languages.
3. We introduce many previously unpublished optimisations and techniques developed for the different K.U.Leuven JCHR Systems (Chapter 7), and show how we have significantly improved or refined existing ones.
4. We show how CHR's standard execution scheme and optimisations can be adapted—without sacrificing on efficiency—to deal with recent expressive language features, such as batch semantics, rule priorities, and negation as absence. Our approach for adding priorities builds further on the foundational work by De Koninck (2008).

Overview In Section 8.1, we first specify a restricted, normalised core language used in the remainder of the chapter. Then, Section 8.2 introduces the basic compilation methodology, including the principal data structures and operations, and a basic, naive compilation scheme. In Section 8.3, we gradually add the more advanced analyses and optimisations used by a state-of-the-art CHR compiler. Throughout, we employ a tutorial-style presentation, using high-level pseudocode and many clarifying examples. We empirically evaluate our approach in Section 8.4, and compare it with related work in Section 8.5. To conclude, Section 8.6 overviews ongoing research, and some promising directions for future research.

8.1 Core language and normal form

In this chapter, we essentially consider a subset of CHR₂ extended with negation as absence. This corresponds more or less to what is currently implemented by JCHR₂. We will abstract from aspects related to the integration with the Java host language; those were covered in sufficient detail earlier in Chapter 7. Instead, we focus on demonstrating the fundamental aspects of our compilation methods for the extended CHR₂ language.

For ease of presentation, we consider only a restricted core language. When relevant, we will explain how to handle other aspects of the full JCHR $\mathcal{2}$ language as well. By default though, we assume the following restrictions:

1. we only consider static priority constraints
2. programs do not explicitly access rule identifiers (Section 5.1.2)
3. no nested negation is allowed

We refer to (De Koninck 2008) for techniques to deal with dynamic priorities.

For convenience, all rules are furthermore assumed to be transformed into a normal form. The normalisation steps taken include:

1. all nested conjunctions are flattened as in Section 5.2.1
2. priority constraints are reduced to CHR^{FP}-style static priority numbers
3. all rule heads are linearised as discussed in Section 4.1.3

To transform static priority constraints into static priority numbers, we use a standard topological sorting algorithm (see e.g. Cormen et al. 2009, §22.4). From Section 5.2.3 we know this is always possible, but also that this process potentially introduces many unspecified priority constraints. Researching how to exploit the increased freedom offered by the priority constraints—starting with the choice between alternative linear extensions—is part of future work. Because it better matches with the implementation, we let a lower number denote a higher priority, exactly as in CHR^{FP}.

Example 8.1. The following rule occurs in the CHR $\mathcal{2}$ version of the RAM simulator by Sneyers et al. (2009). It is used as a running example in the upcoming sections (for a brief introduction to the RAM handler: see Example 5.1).

$$\begin{array}{l} \text{add } @ \text{ -pc}(L), \text{ +prog}(L, \text{ADD}, B, A), \\ \text{ -mem}(A, X), \text{ +mem}(B, Y) \Rightarrow \text{ mem}(A, X+Y) \ \& \ \text{ pc}(L+1). \end{array}$$

Using ‘::’ to denote the priority number, the rule’s normal form is as follows:

$$\begin{array}{l} \text{add} :: 0 \ @ \\ \text{ -pc}(L), \text{ -mem}(A, X), \\ \text{ +prog}(L_1, I, B, A_1), \text{ +mem}(B_1, Y), \\ \text{ ?}(L_1 = L, I = \text{ADD}, B_1 = B, A_1 = A) \\ \Rightarrow \\ \text{ mem}(A, X+Y) \ \& \ \text{ pc}(L+1). \end{array}$$

In normalised rule heads, all occurrences are linearised—that is, all guards are made explicit—and rule conditions of the same kind are grouped together—first all removed occurrences, followed by the kept ones,¹ then the guards, and finally the similarly linearised negated conjunctions (if any). All rule bodies, are sequential conjunctions of batch conjunctions (cf. Section 5.2.1).

¹In CHR’s syntax (and its conventional normal form), the kept occurrences precede the removed ones. CHR compilers though consider removed occurrences first (cf. Section 8.3.4.1).

8.2 Basic Compilation Methodology

CHR programs are typically statically compiled to host-language code. This allows the generation of specialised, highly optimised code. In Section 8.2.2, we introduce the basic compilation scheme, and extend it to deal with negation and priorities in Sections 8.2.3 and 8.2.4. This scheme is still fairly naive and inefficient. It is designed, however, to be clear, and obviously correct with respect to the ω_2 semantics of Section 5.2. In Section 8.3, we gradually transform it into a highly optimised, efficient compilation scheme.

Before introducing the compilation scheme, we first describe the principal data structures and operations it uses in Section 8.2.1.

8.2.1 Principal data structures and operations

8.2.1.1 The constraint store

The central data structure is the *CHR constraint store*. Efficient storage and retrieval of constraints is crucial for performance. The actual implementation of the constraint store (and its indexes: see Section 8.3.2.2) is outside the scope of this section. Instead, we simply list its basic properties and operations.

As in most CHR operational semantics, each constraint is assigned a unique constraint identifier, an increasing integer number that may also serve as a timestamp. In the pseudo-code used throughout this chapter, a CHR constraint is denoted as $c(\bar{X})\#ID$. We often make the transition from an identifier ID to its corresponding constraint implicitly, much like a pointer in imperative languages. The basic constraint store operations are the following:

- `create($c/n, \bar{X}$)` creates a new constraint of given predicate c/n with n arguments \bar{X} , and returns its unique identifier
- `store(ID)` adds the referenced constraint (created earlier with the `create` operation) to the constraint store
- `remove(ID)` removes the constraint from the constraint store data structures; subsequent calls of `alive(ID)` return false
- `alive(ID)` tests whether the corresponding constraint is alive, that is: not yet removed by the `remove` operation
- `lookup(c/n)` Returns an iterator (see below) over all stored constraints of predicate c/n

8.2.1.2 Constraint iterators

To iterate over (subsets of) the constraint store, we use the well-known *iterator* abstraction (see e.g. Gamma et al. 1995). Even though every CHR implementation

relies on some form of iterators, we have been the first to explicitly fix their necessary requirements (Van Weert et al. 2008). We require the iterators returned by `lookup` operations to have at least the following four properties:

1. *robust*: even if constraints are added or removed while a constraint iteration is interrupted, iteration can be resumed without failure
2. *correct*: iterators only return live constraints
3. *complete*: all constraints that were stored at the moment of the iterator's creation are returned at least once (if still live)
4. *weakly duplicate-free*: an uninterrupted iteration does not contain duplicates; if the constraint store is modified while iteration is suspended, constraints returned prior to that suspension may be returned once more

A preferred property is moreover that iterators are *strongly duplicate-free*, which entails that they never return a constraint more than once.

Iterators offered by predefined data structures typically do not have all required properties. Iterators returned by the standard Java data structures, for instance, are not robust under modifications (Bloch et al. 2010). The following fragment is found in the API specifications of these Java classes:

“The iterators returned [...] are *fail-fast*: if the [underlying data structure] is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. ”

There are at least two naive approaches to implementing robustness:

1. Before the iteration starts, create a complete list of (references to) all current constraints. This dedicated copy is only used by the iterator, and is thus never structurally modified during iteration.
2. After an interruption, if a modification has occurred, restart the iteration from scratch (thus sacrificing strongly duplicate-freeness).

Both approaches result in obvious inefficiencies. An efficient iterator implementation returns any first or next constraint in (amortised) constant time. Efficiently implementing a robust constraint iterator can be hard, depending on the underlying data structures (cf. Section 8.3.2.2).

8.2.1.3 The propagation history

The propagation history is a set of rule instances, used to prevent unwanted reapplication. It supports the two obvious operations `addToHistory(rule, $\overline{\text{ID}}$)` and `inHistory(rule, $\overline{\text{ID}}$)`, where `rule` denotes a rule's unique identifier, and $\overline{\text{ID}}$

a sequence of constraint identifiers. A third operation, `cleanHistory()`, removes all rule instances that are no longer applicable. History implementation and optimisation is detailed in Chapter 10.

8.2.2 Basic compilation scheme

A central concept in our evaluation strategy is the *active constraint* (see also the refined operational semantics in Section 4.2.3). By default each added constraint is *activated* once; reactivation is discussed later in Section 8.3.5. The active constraint goes through all its occurrences in the program, searching for applicable rule instances. We perform *lazy* matching, i.e., each time an applicable rule instance is found, it is fired immediately. As seen in Section 2.2.3, this constitutes the fundamental difference with *eager* matching algorithms such as Rete and TREAT, which first compute all applicable instances before selecting one to fire. We further discuss related work also in Section 8.5.

The basic `activate` procedure for a constraint c with n positive occurrences has the following form:

```

procedure activate( $c(\overline{X})\#\text{ID}$ )
  if not alive( $\text{ID}$ ) return
  if occurrence_ $c_1$ ( $\text{ID}$ ,  $\overline{X}$ ) return
   $\vdots$ 
  if occurrence_ $c_n$ ( $\text{ID}$ ,  $\overline{X}$ ) return
end

```

By default, an active constraint traverses its occurrences in a top-down, left-to-right order (later sections refine this *occurrence order*). An *occurrence procedure* returns **false** if the active constraint is still alive after firing all applicable rule instances matching that occurrence, and **true** otherwise.

We now introduce the basic compilation scheme for each individual occurrence procedure `occurrence_ c_i` . Our presentation assumes a rule in the following generic normal form (cf. Section 8.1 for the normalisation process):

$$\begin{aligned}
 \rho :: p \text{ @} \\
 & -c_1(\overline{X}_1), \dots, -c_r(\overline{X}_r), +c_{r+1}(\overline{X}_{r+1}), \dots, +c_n(\overline{X}_n), \\
 & ?g_1, \dots, ?g_\gamma, \sim N_1, \dots, \sim N_m \\
 \Rightarrow & b_1(\overline{Y}_1) \ \& \ \dots \ \& \ b_\delta(\overline{Y}_\delta).
 \end{aligned}$$

with $\sim N_i = \sim(c_{i,1}(\overline{X}_{i,1}), \dots, c_{i,n_i}(\overline{X}_{i,n_i}), ?g_{i,1}, \dots, ?g_{i,\gamma_i})$. All \overline{X} s denote sequences of mutually distinct variables, all b s are built-in or CHR constraint predicates, ρ a unique identifier, and p a static priority number à la CHR^{IP}. For ease of presentation, we only consider bodies consisting of a single batch conjunction here. Sequential conjunctions are dealt with in Chapter 9.

Listing 8.1 The basic compilation scheme for the j -th positive occurrence of constraint c_i , an occurrence $\pm c_i(\overline{X}_i)$ in a rule ρ in normal form (with $m = 0$).

```

1  procedure occurrence_ $c_i$ _j( $ID_i, \overline{X}_i$ )
2    foreach  $c_1(\overline{X}_1)\#ID_1$  in lookup( $c_1$ )
3      ..
4        foreach  $c_{i-1}(\overline{X}_{i-1})\#ID_{i-1}$  in lookup( $c_{i-1}$ )
5          foreach  $c_{i+1}(\overline{X}_{i+1})\#ID_{i+1}$  in lookup( $c_{i+1}$ )
6            ..
7              foreach  $c_n(\overline{X}_n)\#ID_n$  in lookup( $c_n$ )
8                if alive( $ID_1$ ) and ... alive( $ID_n$ )
9                  if allDifferent( $ID_1, \dots, ID_n$ )
10                     if  $g_1$  and ... and  $g_\gamma$ 
11                       if not inHistory( $\rho, ID_1, \dots, ID_n$ )
12                         addToHistory( $\rho, ID_1, \dots, ID_n$ );
13                         remove( $ID_1$ ); ...; remove( $ID_r$ );
14                          $ID_1^a = \text{create}(b_1, \overline{Y}_1)$ ; store( $ID_1^a$ );
15                         ..
16                          $ID_\delta^a = \text{create}(b_\delta, \overline{Y}_\delta)$ ; store( $ID_\delta^a$ );
17                         cleanHistory();
18                         activate( $ID_1^a$ ); ...; activate( $ID_\delta^a$ );
19                         if not alive( $ID_i$ ) return true
20                       end
21                     end
22                   ..

```

Listing 8.1 shows the basic compilation of a single occurrence c_i in a rule without negation. We further assume the body only contains CHR constraints. Adding built-in constraints or even arbitrary host language statements is not hard. The problem of efficiently reactivating CHR constraints is examined later in Section 8.3.5; until then we mostly ignore this aspect of CHR execution.

Lines 2–7 constitute a nested iteration over all $n - 1$ *join partners*, i.e. constraints that may match the remaining positive occurrences. If the active constraint fixes some variables shared with the remainder of the head, it is said to *seed* the search for matching partners. A rule instance is valid if all its constraints are alive (line 8) and mutually distinct (line 9), and all guard constraints are satisfied (line 10). After verifying that it has not fired before (line 11), the rule instance is fired: the history is updated (line 12), the necessary constraints are removed (line 13), and the different batch conjunctions from the body are executed (lines 14–24). After the nested loops (not shown) follows a ‘**return false**’ statement, signalling the next occurrence procedure must be called.

Listing 8.2 Naive compilation of the (removed) `pc(L)` occurrence of the RAM simulator rule of Ex. 8.1.

```

1  procedure occurrence_pc_1(ID1,L)
2    foreach mem(A,X)#ID2 in lookup(mem)
3      foreach prog(L1,I,B,A1)#ID3 in lookup(prog)
4        foreach mem(B1,Y)#ID4 in lookup(mem)
5          if alive(ID1) and alive(ID2) and alive(ID3) and alive(ID4)
6            if allDifferent(ID1, ID2, ID3, ID4)
7              if L1 = L and I = "add" and A1 = A and B1 = B
8                if notInHistory(add, ID1, ID2, ID3, ID4)
9                  addToHistory(add, ID1, ID2, ID3, ID4);
10                 remove(ID1); remove(ID2);
11                 ID1a = create(mem, A, X+Y);
12                 store(ID1a);
13                 ID2a = create(pc, L+1);
14                 store(ID2a);
15                 cleanHistory();
16                 activate(ID1a); activate(ID2a);
17                 return true
18             end
19         end
20     end

```

Example 8.2 (Running example). The naive compilation of the `pc` occurrence of Example 8.1—the first occurrence of `pc/1` in the RAM handler—is shown in Listing 8.2. This example is used as a running example later in Section 8.3.

Added constraints are activated left-to-right, each traversing occurrences in a top-down, left-to-right order. This is still more or less in line with the conventional refined operational semantics of CHR, and leads to a clean, natural default operational behaviour. A crucial difference though with ω_r -based execution is that constraints are only activated (line 18) after all other constraints have been created and added to the store (lines 14–16), as is required by the ω_2 (and ω_p) semantics of batch conjunction. In traditional ω_r -based CHR systems, each constraint is activated immediately after it is created (and stored). As said earlier, ω_r -like sequential conjunctions are dealt with in Chapter 9.

The correctness of the scheme heavily relies on the properties of the constraint iterators we established in Section 8.2.1.2. Their robustness property ensures suspended iterations can always be resumed, even if meanwhile the underlying constraint store has been altered. Completeness guarantees all necessary candidate partner constraints are tried at least once. Partners added after an iteration

Listing 8.3 The basic compilation scheme for the i 'th negated conjunction of rule ρ in normal form. Apart from line 8, this is completely analogous to the positive case (Fig. 8.1).

```

1  procedure negated_ρ_i(ID1, ..., IDn, X̄1, ..., X̄n)
2    foreach ci,1(X̄i,1)#ID1n in lookup(ci,1)
3      ⋮
4      foreach ci,ni(X̄i,ni)#IDnin in lookup(ci,ni)
5        if alive(ID1) and ... alive(IDn)
6          if allDifferent(ID1, ..., IDn, ID1n, ..., IDnin)
7            if gi,1 and ... and gi,γi
8              return true
9            end
10       ⋮

```

started must not be included, because these have already been active themselves. The duplicate-freeness property finally, avoids trivial non-termination in the nested loops. For weakly duplicate-free iterators, the propagation history will intercept any duplicate rule instances found.

8.2.3 Extension with negation

Adding negation to the basic compilation scheme is relatively straightforward. Only two extensions are required. Firstly, for each negated conjunction, a test of form

$$\mathbf{if\ not\ negated_}\rho_i(\text{ID}_1, \dots, \text{ID}_n, \bar{X}_1, \dots, \bar{X}_n)$$

is added between lines 10 and 11 of Fig. 8.1 ($1 \leq i \leq m$). These *negation conditions* ensure the constraint store does not contain constraints matching the negated conjunctions. The basic compilation scheme for these procedures is shown in Fig. 8.3. The arguments passed are the identifiers and variables of the constraints matched in the positive part of the head.

Secondly, removed constraints must be activated, since removing constraints may cause new rule instances to become applicable. Suited calls to the obvious `activate($\sim c(\bar{X})\#\text{ID}$)` procedures are thus added to the body evaluation, which in turn use `neg_occurrence_ci,j-k(ID, X̄i,j)` procedures generated for each *negative occurrence* $c_{i,j}$. The main difference with the positive case of Listing 8.1 is that all n positive occurrences must be matched. The removed constraint's arguments can be used to seed the join though.

The basic compilation scheme in this section corresponds closely to the lazy matching algorithm LEAPS (cf. Section 2.2.3). The main difference is the way reapplication is prevented. We discuss this further in Sections 8.5 and 10.5.1.

8.2.4 Extension with priorities

Priorities add the restriction that no rule instance may fire if any instance of higher priority is applicable. Consequently, an active constraint should only consider an occurrence after all other constraints have tried all their occurrences of higher priority. For this, we introduce a final runtime data structure called the *schedule*. The items on the schedule are called *continuations*, and represent added or removed constraints that still have to be activated starting from some priority. The schedule—technically a *priority queue*; see e.g. Cormen et al. (2009, §6.5)²—efficiently supports the following operations:

- `schedule(\mathcal{X})` adds a continuation \mathcal{X} of form $c(\overline{X})\#ID@p$ or $\sim c(\overline{X})\#ID@p$ to the schedule
- `pollScheduled()` removes and returns a next scheduled continuation with minimal priority number p
- `scheduledPriority()` returns the minimal priority number p of all scheduled continuations, or $+\infty$ if the schedule is empty

We assume $c(\overline{X})\#ID@p$ continuations are automatically removed from the schedule when or after the $c(\overline{X})\#ID$ constraint is killed.

Removed and added constraints are thus no longer activated directly, but simply scheduled at their highest priority. In other words, all calls of `activate` (e.g. lines 18 and 24 of Listing 8.1) are replaced with suited calls of `schedule(\mathcal{X})`. After all batch conjuncts are scheduled, the following procedure is called, with p_{active} the priority of the applied rule:

```

procedure activate( $p_{active}$ )
  while scheduledPriority() <  $p_{active}$ 
    activate(pollScheduled(),  $p_{active}$ );
  end
end

```

This calls updated versions of the `activate` procedures of Section 8.2.2, which now take the form shown in Listing 8.4; the scheme for removed constraints is analogous. Using a `switch` statement to start at the right priority, the active constraint keeps trying occurrences³ until it is either removed, or until it has to yield control to the previously active constraint or to some other constraint on the schedule. The occurrence order is changed to reflect priorities. For occurrences of the same priority, the familiar top-down order is used by default.

²For static priorities an efficient schedule is easily obtained as a straightforward array of lists, one for each static priority number. For dynamic priorities more complex priority queue data structures are required, as discussed by De Koninck (2008).

³Our `switch` statement's fall through mechanism ensures that, if no `return` is executed, occurrences of lower priority are tried as well.

Listing 8.4 Activation of c constraints, extended to deal with priorities. We assume occurrences f_i to l_i have priority p_i (that is: $l_i = f_{i+1}$), with $p_{i+1} < p_i$.

```

1  procedure activate( $c(\overline{X})\#ID@p$ ,  $p_{active}$ )
2      switch  $p$ 
3          case  $p_1$ 
4              ...
5          case  $p_i$ 
6              if occurrence_ $c_{f_i}(ID, \overline{X})$  return
7                  :
8              if occurrence_ $c_{l_i}(ID, \overline{X})$  return
9              if  $p_{i+1} \geq p_{active}$  or  $p_{i+1} > \text{scheduledPriority}()$ 
10                 schedule( $c(\overline{X})\#ID@p_{i+1}$ )
11                 return
12             end
13         case  $p_{i+1}$ 
14             ...
15     end
16 end

```

For now, we opted to follow De Koninck (2008), and do not consider occurrences of the same priority as a previously active constraint. In other words, the following invariant holds: *at most one constraint is active at any given priority*. For single-batch rule bodies, this is allowed by ω_2 : the semantics only requires all rule instances of strictly higher priority to fire, and intentionally remains nondeterministic with respect to rules of equal (or incomparable) priority. While the above invariant simplifies certain analyses and optimisations, it arguably results in a somewhat irregular runtime behaviour (for instance for recursive rules). We return to this subtle design choice in Chapter 9.

8.3 Program Analysis and Optimisation

This section provides a comprehensive compendium of CHR program analyses and compiler optimisations, applied to the basic compilation scheme introduced in Section 8.2. Clearly, it is not possible to discuss all techniques in full detail. We therefore place the many contributions in one unified, accessible framework, and refer to relevant literature for more detailed information.

A main goal is to divulge the lessons learned and experience gained while developing the different K.U.Leuven JCHR systems (Section 7.3), and to highlight several innovating contributions made in this context. These range from extensions

of existing techniques to CHR² language features, over improved optimisations facilitated by the imperative host language, to completely new optimisation techniques applicable in principle in any CHR system. As a quick visual aid, we have annotated each subsection with a box, indicating whether or not that particular compiler optimisation is implemented in JCHR and JCHR², and to which extend the presented content is new.

Example 8.3. An example box would be **extended**, which would indicate that that particular technique is implemented in JCHR² (the checked second box), but not in the original JCHR system (the unchecked first box). The label used in the (optional) third box denotes our main contribution in the context of the optimisation at hand. Labels used are:

extended We extended that particular optimisation idea to either priorities or negation as absence.

improved We (considerably) improved the optimisation.

first We were the first to introduce this technique, which has since been incorporated by other system.

new Like **first**, but to the best of our knowledge this optimisation is currently exclusively performed by our JCHR systems.

8.3.1 Constraint invariants

In Sections 5.1.6–5.1.7, we introduced set semantics and functional dependencies, two very important common constraint invariants. These invariants are not only important for the programmer though, they are also invaluable information for a CHR compiler. Knowing about constraint invariants facilitates, among other things, optimal indexing (Section 8.3.2.3) and join ordering (Section 8.3.2.7), both decisive for the runtime complexity of CHR programs.

8.3.1.1 Deriving constraint invariants



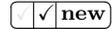
Most CHR systems do not support invariant declarations such as those proposed for CHR². Instead, it is common practice in CHR programming to specify constraint invariants by adding the appropriate rules (as discussed at length in Sections 5.1.6–5.1.7). It therefore remains very important that CHR compilers are capable of deriving constraint invariants from such rules. In fact, they may even be derived from rules that enforce more specific invariants:

Example 8.4. The following rule adapted from the Fibonacci heap program by Sneyers et al. (2006a) implies **min** is a singleton constraint:

```
keep_min @ +min(_,A), -min(_,B), A ≤ B.
```

For detailed descriptions of algorithms used to derive constraint invariants, we refer to (Holzbaur et al. 2005; Duck 2005; Duck and Schrijvers 2005).

8.3.1.2 Unenforced constraint invariants



Unavoidably, enforcing constraint invariants involves a runtime overhead: both for set semantics and functional dependencies, the constraint store has to be checked for matching constraints. As discussed in Section 8.3.2.2, specific index data structures are required to efficiently perform these operations. In the best case, these indexes are useful for join computations as well, but often indexes have to be created specifically for the efficient enforcement of constraint invariants.

However, very often the programmer knows:

- a) the user queries are duplicate-free
- b) the program never produces duplicate constraints

Up to a point, for specific cases, the latter property could be derived from the program by the compiler. In general though such an analysis is hard (no doubt even undecidable). We therefore provide specific variants for our constraint invariant declarations for which no runtime tests are generated. Consequently, no redundant, relatively expensive indexing structures must ever be created just to enforce invariants that are known to hold.

Example 8.5. In JCHR2's current syntax, the constraint declarations for the RAM program shown earlier in Example 5.16 can safely be changed to:

```
constraint pc(int label) # *singleton,
           mem(int addr, int val) # *key(addr),
           prog(int label, int, int, int) # *key(label);
```

While not affecting the operational semantics, this knowledge can be exploited by the compiler. These annotations are also invaluable as part of a program's documentation, and could be verified automatically if desired, either when running the program in a debugging mode, or maybe someday even by static program verification. Statically verifying e.g. that the RAM program maintains these invariants, given a correct query, would be straightforward.

8.3.2 Optimising join computation

The most critical part of any rule-based system is the search for matching partner constraints to form rule instances. This section surveys several techniques to optimise this process, called *join computation*. Over the course of this section, it should become clear that the effectiveness and applicability of most of these optimisations is largely determined by the order in which partner constraints are looked up. The problem of finding the optimal *join ordering* is discussed last in Section 8.3.2.7. Until then, we assume all partner constraints are looked up in a fixed, left-to-right order.

Listing 8.5 Optimised compilation of the RAM simulator example of Listing 8.2 after loop-invariant code motion.

```

1 procedure occurrence_pc_1(ID1,L)
2   foreach mem(A,X)#ID2 in lookup(mem)
3     foreach prog(L1,I,B,A1)#ID3 in lookup(prog)
4       if L1 = L and I = "and" and A1 = A
5         foreach mem(B1,Y)#ID4 in lookup(mem)
6           if ID2 ≠ ID4 and B1 = B
7             ..

```

We mostly explain and illustrate join computation optimisation for the positive part only (Listing 8.1 extended with negation conditions). All techniques surveyed though equally apply to joining negated conjunctions (Listing 8.3).

8.3.2.1 Loop-invariant code motion

✓✓ extended

All tests, particularly identifier comparisons and most guards, should normally be performed as early as possible. The following example clarifies this:

Example 8.6. The improved compilation of the RAM simulator example introduced in the previous section is listed in Listing 8.5. Moving the ‘L = L₁’ guard to line 4, for instance, avoids enumerating all `mem(B1,Y)` memory cells before the right program instruction is found.

For kept occurrences, if after the body’s execution the search for join partners is resumed⁴, hoisting a test is only correct if the body cannot change the outcome of the test. These tests are called *loop-invariant*. If tests that are not loop-invariant are hoisted, they may not be retested after the execution of the body, in which case inner loops may cause rule instances to fire for which these tests no longer hold. In pure CHR, all guards are monotonic, and are thus always loop-invariant. For negation conditions, non-pure guards, and liveness tests, however, the compiler cannot a prio assume they are loop-invariant:

- A negated condition is loop-invariant if the body never adds, either directly or indirectly, constraints that match that condition.
- Non-monotonic guards are treated similarly to negated conditions.⁵

⁴The search is also never resumed—that is, besides when the active occurrence is a removed occurrence—if none of the lookups return an iterator: see Section 8.3.2.3.

⁵Often the compiler has insufficient knowledge on arbitrary host language statements to properly reason on loop-invariance. JCHR2 therefore offers the following meta data facilities:

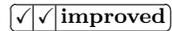
- Java methods can be declared to be monotonic (using e.g. annotations)
- individual guards can be declared loop-invariant (using a pragma)

- A liveness test is loop-invariant if the corresponding matched constraint cannot be (indirectly) removed by firing the rule. As iterators only return live constraints (cf. Section 8.2.1), such liveness tests can simply be omitted.

To determine loop-invariance, static control flow analysis is used, e.g. based on some form of abstract interpretation. Techniques to deal with tests that are not loop-invariant, or for which loop-invariance cannot be proven, are discussed later in Section 8.3.2.5.

Caveat Moving a test to an outer loop may increase the number of times it is called (e.g. if inner iterations are frequently empty). Unfortunately, for computationally more expensive tests such as negation condition or even expensive guards, loop-invariant code motion may thus unwittingly decrease performance.

8.3.2.2 Constraint indexing



Efficient, selective lookups of candidate join partners are of paramount importance. Therefore, *indexes* on the constraint store are used. Essentially, this incorporates one or more guards into the lookup operation, such that it (very efficiently) returns only those constraints that satisfy these guards.

Example 8.7. In Listing 8.5, line 3 iterates over all `prog` constraints, each time immediately testing the ‘ $L_1 = L$ ’ guard. The constraint’s invariants, however, imply there is at most one `prog` constraint with given label `L` (see Examples 5.16 and 8.5). Retrieving this single constraint using an appropriate index reduces the time complexity of this part of the join computation from linear to constant. A similar reasoning applies to the lookup of the `mem` constraint (lines 5–6).

We now very briefly discuss indexing techniques used in JCHR and other current CHR systems. For a more extensive overview, we refer for instance to (Sneyers 2008, Chapter 7).

For lookups of join partners via one or more known arguments, tree-, hash-, or array-based indexes are used (Duck 2005; Holzbaur et al. 2005; Schrijvers 2005; Sneyers et al. 2006a). Tree-based indexes cannot only be used for equality-based lookups, but also for pruning the join partner search space in case of inequality guards (Duck 2005). The other two types are particularly interesting as they offer (amortised) constant time operations.

Because constraint indexes are used extensively, finetuning these data structures is very important. It also pays to implement specialised indexes for common cases. JCHR e.g. uses specialised data structures for integer and finite domain (enums, booleans, ...) arguments, never-removed constraints, etc.

For technical reasons, these indexes are commonly only used for ground arguments. JCHR was the first system to also allow hash indexing on non-ground variables, using an observer-based approach to maintain data structure consistency. De Koninck (2008) has since used this approach for CHR^{FP} as well.

Listing 8.6 Our running example using indexing, and exploiting functional dependency and set semantics invariants.

```

1 procedure occurrence_pc_1(ID1, L)
2   foreach mem(A, X)#ID2 in lookup(mem)
3     prog(L1, I, B, A1)#ID3 = lookup_s(prog, {L1=L})
4     if ID3 ≠ nil and I = "and" and A1 = A
5       mem(B1, Y)#ID4 in lookup_s(mem, {B1=B})
6       if ID4 ≠ nil and ID4 ≠ ID2
7         ..

```

The indexing technique for unbound logical variables commonly used by CHR(Prolog) implementations is *attributed variables* (Holzbaur and Frühwirth 1999; 2000a). With this technique, variables contain references to all constraints in which they occur. This allows constant time lookups of partner constraints via shared variables. Inspired by this concept, Sarna-Starosta and Schrijvers 2008a devised an improved internal representation of CHR(Prolog) argument values that constitutes an efficient alternative to standard indexing.

While indexes often improve performance substantially, even result in better runtime complexity, each index also involves a non-negligible maintenance cost. Current CHR systems generally eagerly introduce indexes whenever possible. If indexes are not or only scarcely used, or constraints are continuously added and removed, indexes may therefore even have a negative impact on performance. Section 8.3.3 discusses techniques to reduce indexing overhead.

8.3.2.3 Exploiting constraint invariants

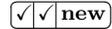


Firstly, if (derived) invariants imply a lookup returns at most one constraint, more efficient index structures can be used, and specialised lookup routines that return a single constraint instead of an iterator (Holzbaur et al. 2005). We call such specialised procedures *singleton lookups*, and denote them `lookup_s`.

Example 8.8. Listing 8.6 shows the optimised compilation of the RAM example. If the specialised singleton `lookup_s` operations on lines 3–4 use proper indexing, both partners are found in $\mathcal{O}(1)$ time, instead of $\mathcal{O}(p \times m)$, with p the number of RAM program instructions, and m the number of used memory cells. Also, the functional dependency of the `prog` constraint was used to select optimal indexing (line 3). When unaware of this invariant, the compiler would typically add a redundant index on the combination of the first, second and fourth arguments.

A second, equally important invariant-based optimisation is hence that, by reducing the number of maintained indexes, functional dependencies tend to improve both space and time performance considerably (Holzbaur et al. 2005).

8.3.2.4 Pre-commit backjumping



We say a constraint in lines 1–7 of Listing 8.1 (i.e., the active constraint or some join partner) *seeds* either a guard, a negation condition, or a partner constraint lookup, if it binds variables required to execute the latter.

Example 8.9. The following rule head is taken from the WALTZ program:

```
initial_boundary_junction_L @
  -stage("find initial boundary"),
  +junction(P1, P2, _, Base, "L"),
  -edge(Base, P1, Q1, _, Plotted1),
  -edge(Base, P2, Q2, _, Plotted2),
  ~(junction(_, _, _, B, _), B > Base)
⇒ ...
```

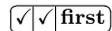
Suppose the partners of an active `stage` constraint are matched in the order they appear in the head. Then the only constraint seeding the negation condition is the one matched by the `junction` occurrence. Therefore, when the negation condition fails, resuming the nested iteration over `edge` constraints is pointless. Instead, the next `junction` constraint should be tried.

The *backjumping* optimisation ensures that when a guard or negation condition fails, or when a partner constraint lookup returns an empty iterator, a jump is executed to resume the *inner-most seeding loop*. This loop corresponds to the last seeding partner constraint in the join order. Only if this partner is retrieved using a singleton lookup, the loop immediately surrounding that lookup is taken. If there is no inner-most seeding loop—either because there is no seeding partner, or because all partners prior to and including the inner-most seeding partner have set semantics—`true` is simply returned.

Without backjumping, a phenomenon called *trashing* occurs, where inner loops are iterated exhaustively, only to find the test or lookup always fails.⁶ Analogous optimisations are used in constraint solvers and logic programming.

JCHR2 is currently the only system to implement this form of backjumping during the search for partner constraints. Particularly for rules with many heads, the potential performance gains of backjumping are significant.

8.3.2.5 Post-commit backjumping



Section 8.3.2.1 discussed loop-invariant code motion, and some of its limitations. We now show how a second form of backjumping allows tests that are not loop-invariant to be moved into outer loops nevertheless. The concepts trashing, backjumping, and seeding loop were introduced in Section 8.3.2.4.

⁶In the case of Example 8.9, with proper indexing, trashing would actually be limited, as by functional dependency only a single matching `edge` constraint exists for each inner loop.

Liveness tests Resuming search after a constraint matched by an outer loop is removed—by the rule itself or indirectly by some activated constraint—is a clear source of trashing. The solution is to move those liveness tests of line 8 of Listing 8.1 that may change by evaluating the body to after line 19, that is: *after* the evaluation of the body. The liveness of the outermost loop’s constraint is tested first, and so on. If one of the partners tests dead, a backjump is used to the corresponding loop. If the dead partner is a singleton partner, the inner-most seeding loop is used instead, or `true` is returned if no such loop exists. Of course, if the rule itself removes some partner, the jump becomes unconditional.

Negation conditions and guards Similarly, a non-invariant negation condition or guard may be hoisted into outer loops as long as it is retested after each commit. If the test fails, a jump to its inner-most seeding loop is performed, or `true` is returned if no such loop exists.

Unlike liveness tests and most guards, testing negation conditions can be expensive. It therefore pays to detect whether the body always adds constraints matching the condition. In this case, the backjump becomes unconditional.

Example 8.10. The following rule in the MANNERS program is a classic example of a common pattern in rule-based programs with negation:

```
make_path @ ..., ~path(Id, N1, _) => path(Id, N1, S);
```

Before something is added to the store, an often more general pattern (though not always) is used to check something similar is not already present. In such cases retesting the negated head is often pointless.

To detect these cases, the analysis used to detect loop invariance as sketched in Section 8.3.2.1) is readily extended.

Post-commit backjumping was first introduced for JCHR (Van Weert 2005), and was later incorporated into CCHR as well (Wuille 2007). CHR(Prolog) systems on the contrary use standard backtracking search, i.e. without pre- or post-commit backjumping, and are therefore more prone to trashing effects.

8.3.2.6 Fragile iterators

✓✓improved

Due to the highly dynamic nature of the constraint store, the robustness property of iterators is hard to implement efficiently (see Section 8.2.1.2). We therefore distinguish so-called *fragile iterators*. These are leaner, more efficient iterators without any extra tests and facilities required for robustness. Depending on the underlying index structure, such iterators can be implemented with a superior complexity (cf. Section 8.2.1.2 for potential inefficiencies due to robustness). But even if guaranteeing robustness involves only an (amortised) constant overhead, the performance gains are significant, since iterators are typically very heavily used during a program’s execution.

Fragile iterators though should be handled with due care, as they may fail or produce incorrect results if resumed after structural modifications to the store. To preserve correctness, fragile iterators are thus only used in the following cases:

1. If after the execution of the body the iterator is never resumed, e.g. if the active constraint is removed, or due to an unconditional backjump to a more outer loop.
2. If static control flow analysis shows the body is guaranteed not to modify the relevant part of the constraint store. Certain iterators, moreover, still function correctly as long as only constraints are added. In this case, only the possibility of constraint removals should be analysed.⁷
3. None of the iterators used to test negated conditions (Listing 8.3) have to be robust.

Example 8.11. In the RAM running example, all iterators used may be fragile as the rule removes the active constraint.

Certain CHR(Prolog) systems distinguish *universal* and *existential* join computations for kept and removed active occurrences respectively (Holzbaur et al. 2005; Schrijvers 2005). The more fine grained application of fragile iterators, however, is pioneered by the imperative CCHR and JCHR systems.

8.3.2.7 Join ordering

✓✓ improved

Because the effectiveness of all optimisations in this section is determined by the order in which join partners are looked up, a program's time complexity is often determined by the *join order*. It determines the earliest position where guards, constraint indexes, functional dependencies, etc. can be used to prune the search space. The general goal behind *join ordering* is to maximise this pruning, in order to minimise the number of join partners tried. In general, join ordering is NP complete. The optimal join order may moreover depend on dynamic properties, such as the size of the constraint store, the selectivity of guards, etc. If no functional dependencies are declared or derived, a compiler must rely on heuristics to determine the join order.

Example 8.12. Line 2 of Listing 8.6 iterates over all `mem` constraints. Lacking any information on `A` (or `L`), no index can be used. Using the join order depicted in Listing 8.7, however, all lookups become optimal. For determining this join order, functional dependencies are indispensable (cf. Example 8.5).

⁷The removal of constraints that have already been iterated over is typically not a problem. JCHR exploits this property to allow fragile iterators even for rules that remove the corresponding partner constraint.

Listing 8.7 Optimal join computation for our running example.

```

1 procedure occurrence_pc_1(ID1,L)
2   prog(L1,I,B,A1)#ID3 = lookup_s(prog,{L=L1})
3   if ID3 ≠ nil and I = "and"
4     mem(A,X)#ID2 = lookup_s(mem,{A1=A})
5     if ID2 ≠ nil
6       mem(B1,Y)#ID4 = lookup(mem,{B1=B})
7       if ID4 ≠ nil and ID4 ≠ ID2
8         .

```

To date, De Koninck (2008, Chapter 6) and Sneyers (2008, Chapter 9)⁸ provide the most comprehensive treatment of join ordering for CHR. They derive a reasonable cost formula for join computations that can be heuristically approximated, either statically or dynamically. They also adapt an efficient algorithm from data base literature that joins a specific, relatively common type of rule heads in $\mathcal{O}(n \log n)$ time.^{9,10}

The work of De Koninck and Sneyers was entirely theoretical though. In JCHR2, we made the first concrete implementation of their proposed approach for static join ordering. We created three distinct join orderers:

1. A simple, naive branch and bound search that exhaustively enumerates all possible join orderings, using the standard branch and bound technique to filter the search space. This join orderer is used only for the simplest rules, as it does not scale at all to rules with many heads.
2. A second, more efficient exhaustive join orderer based on the A^{*} algorithm (Hart, Nilsson, and Raphael 1972). Our algorithm incrementally extends partial joins, occurrence per occurrence, based on heuristics. How an admissible, efficient heuristics can be obtained from the cost function of De Koninck and Sneyers is explained in detail in Appendix A.
3. While the A^{*} join orderer scales reasonably well, join ordering is NP complete, so any exhaustive algorithm is bound to be infeasible in general.

⁸Both chapters are based on their joint workshop paper (De Koninck and Sneyers 2007).

⁹De Koninck (2008) and Sneyers (2008) wrongfully call their algorithm a KBZ algorithm, and credit it to Krishnamurthy, Boral, and Zaniolo (1986). The algorithm they actually describe is the IK algorithm of Ibaraki and Kameda (1984). The core contribution of Krishnamurthy et al. (1986) was that they showed that, in terms of the CHR join order problem, the IK algorithm can be adjusted to compute the join order of all n active constraints in $\mathcal{O}(n^2)$ time instead of $\mathcal{O}(n^2 \log n)$. This is the actual KBZ algorithm.

¹⁰The version of the IK algorithm described by Krishnamurthy et al. (1986), and copied by De Koninck and Sneyers (2007), is not entirely correct. It uses a step where chains of nodes are merged. However, these chains are not necessarily sorted on rank. The normalisation it performs at the root of the merged chain does not solve the problem. The real Ibaraki and Kameda (1984) algorithm correctly normalises (sorts) both chains before merging.

In JCHR2, we found A^{*} is best only used for heads with up to about 8 conjuncts. For larger heads, local search is used, more specifically a random-restart hill climbing algorithm. Our current version corresponds more or less to the ‘iterative improvement’ algorithm of Swami and Gupta (1988).

This combination seems to work well in practice. Still, we only have scratched the surface, and more experimentation is required to determine:

- a) whether the assumptions made by the cost function of De Koninck and Sneyers, and the heuristics used to estimate it, are indeed appropriate.
- b) the right parameters for the local search algorithm (starting points, local moves, stopping criteria, etc.). Moreover, alternative randomised algorithms (genetic algorithms, simulated annealing, etc.) may be better suited; cf. the survey by Steinbrunn et al. (1997).

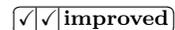
Many more issues must be further investigated, including first-few answers (cf. De Koninck 2008; Sneyers 2008; Bayardo and Miranker 1996), join strategies besides nested-loop joins, a-priori guards besides equality, etc.

The most important open problem though is that, short of reliable estimates for e.g. cardinalities and selectivities, static join ordering frequently produces suboptimal results. In (Van Weert, De Koninck, and Sneyers 2009), we proposed *cardinality annotations* to mend this shortcoming. They allow the user to specify the cardinalities and selectivities of stored constraints at different points during execution. These annotations are not only invaluable to the compiler, but also as program documentation. While these annotations help, they rely on the programmer to supply sufficient and correct information. The only really efficacious solution therefore is dynamic join ordering, as discussed in Section 8.6.1.

8.3.3 Reducing constraint store overhead

Indexes are imperative for the efficient retrieval of candidate join partners. Each index, however, increases the cost of the `store` and `remove` operations, and adds a constant factor overhead in memory consumption. In Section 8.3.2.3, we already saw how functional dependencies are crucial in reducing the number of indexes. Here, we introduce further optimisations to reduce constraint store overhead.

8.3.3.1 Late indexing



Naively, constraints are stored immediately after being created. A constraint’s lifetime, however, is often very short. In fact, a constraint is frequently killed shortly after its activation. The goal of *late storage* (Holzbaur et al. 2005; Schrijvers et al. 2005; Duck 2005; Schrijvers 2005) is to defer storing constraints as long as possible. Consequently, active constraints often get killed without

being stored. This avoids the considerable overhead of constraint store additions and removals. The performance gain is particularly significant if indexes are used.

With late storage, constraints are only stored when they may be *observed*. An active constraint c is observed through an index I if, before execution control returns to c , any other active constraint (positive or negative) causes c to be looked up using I —either as a positive join partner, or while testing a negation condition. The static program analysis that determines when and where constraints must be stored is called the *observation analysis*, and is typically based on abstract interpretation. Late storage is worked out formally in an ω_r -based setting by Schrijvers, Stuckey, and Duck (2005).

Late indexing, first proposed by De Koninck (2008), goes beyond late storage by selectively adding constraints to required subsets of indexes only, instead of all indexes at once. The approach outlined here further refines that of De Koninck (2008), and allows indexing to be postponed (and hence avoided) even more.

In our compilation scheme in Listing 8.1, late indexing postpones the `store` operations of lines 14–16. A constraint is only stored there in those indexes through which it may be observed by negation conditions before it is activated itself. By default, however, a constraint is only indexed while active, when it is about to relinquish control in one of these three cases:

1. prior to the execution of a body conjunction
2. when yielding control at a transition to a lower priority (see Section 8.2.4, Listing 8.4, lines 9–11)
3. after all occurrences have been traversed.

In the first case, the initially activated constraints are obvious. The invariant established in Section 8.2.4 further ensures only rules of strictly higher priority are considered before the previously active constraint regains control.¹¹ In case of a transition from priority p_i to p_{i+1} , any constraint scheduled between p_i and p_{i+1} may be activated¹², and only occurrences of priority higher or equal to p_{i+1} are considered. From this, the observation analysis can accurately determine the required indexing operations.

Our late indexing optimisation may change the order in which rules fire¹³, as active constraints do not always observe not-yet-indexed constraints. All applicable instances are still found though when these constraints are activated themselves. As the ω_r semantics does not allow this, this is another fundamental difference with the conventional late storage optimisation for CHR. It is not yet fully clear when more eager indexing is preferred (cf. Section 8.3.4.4).

¹¹Once sequential conjunctions are added in Chapter 9, this invariant no longer holds, and the observation analysis must be adjusted accordingly.

¹²Actually, added constraints scheduled at p_{i+1} do not have to observe the previously active constraint, as that constraint will also still be activated at p_{i+1} .

¹³Changing the rule order may affect performance, either positively or negatively. This effect can typically be controlled using priorities.

8.3.3.2 Late allocation ☐☐

Since constraints are not always (or never: cf. Section 8.3.4.4) stored, constraint representations do not always have to be created either. Late allocation and late storage are considered separately, because allocation may be required prior to storage. For prioritised programs, for instance, constraints typically have to be added to the schedule (although not always: cf. Section 8.3.4.2).

8.3.3.3 In-place and delayed modifications ☐☐

The *in-place modifications* optimisation by Sneyers et al. (2006b) reuses the original constraint's representation, simply assigning a new identifier, and overwriting the modified arguments. Affected indexes still have to be updated, but indexes independent of the modified arguments require no update.

Example 8.13. Both constraints added by the `add` rule of our running example are modifications of removed constraints. Consequently, lines 10–14 of Listing 8.2 can be replaced for instance by (the constraint and argument declarations of these these constraints are shown in Example 8.5):

```
ID1a = modify(ID2, {value = X+Y})
ID2a = modify(ID1, {label = L+1})
```

The first operation requires no updates to indexes at all because no index exists on the `value` argument of `mem` (cf. Example 8.8).

There are cases where modifications do not occur in the same batch, or even in the same rule. For these cases Sneyers et al. (2006b) developed the so-called *suspension reuse* optimisation. The idea is to keep a pool of recently removed CHR constraints which are also not removed from the index data structures, but simply marked dead. If a modified constraint is then added, the constraint modification can again take place with less overhead.

Of course, suspension reuse itself has a non-negligible overhead. Also, these optimisations may alter the order in which constraints are returned by iterations, which may drastically change a program's performance. Sneyers et al. (2006b) discuss these caveats and trade-offs in more detail.

8.3.3.4 Lazy indexing ☑☐ new

The goal of the JCHR's recent *lazy indexing* optimisation is to only build and maintain those indexes that are actually used. In its simplest form, an index is not built until the first time it is used by a lookup operation. At that point, the runtime traverses all relevant constraints and builds the index (of course at least one constraint list or index is always kept eagerly). In our current implementation, once an index is used and created, newly added constraints are also added to it.

This straightforward optimisation already achieves promising results. We also experimented with more advanced on demand indexing strategies, but further research is required. Full dynamic indexing would e.g. also entail destructing less used indexes, and better runtime heuristics to decide which indexes to create or destruct—e.g. only keep indexes that are ‘sufficiently used’, rather than ‘at least once’ in our current lazy indexing scheme. See also Section 8.6.1.

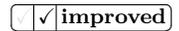
8.3.4 Optimising constraint activation

8.3.4.1 Removal preference



Any CHR compiler uses the following heuristic: when a single rule contains multiple occurrences of the same constraint, by default, an active constraint tries the removed occurrences of that rule first. This is also explicitly required by the refined operational semantics. Removing the active constraint rather than constraints already stored can considerably improve performance (due to late indexing; cf. Section 8.3.3.1), or even avoid non-termination (cf. e.g. Section 5.1.6).

8.3.4.2 Reducing schedule overhead



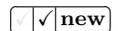
Redundant calls of schedule operations have to be avoided as much as possible. De Koninck (2008) introduced the following optimisations:

1. If firing a rule is known never to activate constraints of higher priority, the call to `activate(p)` can clearly be omitted.
2. This call is also omitted if the active constraint is removed: control always returns to a loop in `activate(p)`, which already checks the schedule for more scheduled activations.
3. Constraints can be activated directly, i.e. without going through the schedule, if their priority is known to be higher than that of the activating rule, and than that of all other constraints activated by that rule.

In JCHR2, we added the following fourth optimisation, further improving the case where the active constraint is removed (the second item in the above list):

4. Instead of returning `true` or `false`, (certain) occurrence methods return the next constraint to activate, or specific default values otherwise. This often saves first adding a constraint to the schedule in a body, only to immediately remove it again in an `activate(p)` loop.

8.3.4.3 Passive removals



Recall from Section 8.2.3 that removed constraints normally become active as well. Of course, if a constraint has no negative occurrences, its removal is not activated.

Listing 8.8 JCHR2 version of the `make_L` rule from the WALTZ program (cf. Listings 2.1–2.3). It contains two opportunities for passive modification.

```

make_L @
  +stage(DETECT_JUNCTIONS),
  -edge(Base, P2, false, L1, Plotted1),
  -edge(Base, P3, false, L2, Plotted2),
  ~edge(Base, _, _, _)
=>
  junction(Base, P2, P3, 0, Junction.L) &
  edge(Base, P2, true, L1, Plotted1) &
  edge(Base, P3, true, L2, Plotted2);

```

We now determine two additional, very common patterns for which the activation of removed constraints may be avoided: *duplicate removals* and *modifications*. The resulting passive removals avoid superfluous traversals of negative occurrences, which in turn leads to more passive negative occurrences—as discussed shortly in Section 8.3.4.4—and improves the results of static control flow analyses. Passive removals thus potentially enable many additional optimisations.

Passive duplicate removals Many heads contain two occurrences, one removed and one kept, that match identical or almost identical constraints. In fact, all rules that enforce set semantics, possibly combined with functional dependency, fall in this category. In these cases, activating the removed constraint is superfluous when static analysis shows all negative occurrences match with the constraint matching the kept occurrence.

Passive modifications Modifications occur frequently, and involve a removed constraint being replaced (often even in the same batch conjunction) with a new, only slightly different version. For modifications, activating the removed (modified) constraint is only necessary in those cases where negative occurrences exist with guards (implicit or explicit) on the modified arguments.

Example 8.14. Listing 8.8 shows the JCHR2 encoding of the WALTZ production rule shown earlier in Listings 2.1–2.3. The removal of the `edge` constraints in this rule can be done passively. The WALTZ program contains no negative occurrence of `edge` that considers the modified `join` argument.

We plan to further generalise the idea of passive modifications as follows. If a modification is not passive, the active removal should still only consider those negative occurrences that contain guards on the modified fields. Similar extensions of the passive duplicate removals are possible as well. This is related to some of the program specialisation techniques discussed in Section 8.3.6.

8.3.4.4 Passive occurrences



extended

An occurrence, positive or negative, is *passive* if static analysis shows the corresponding rule can or must not fire with an active constraint matching it. Passive occurrences are skipped by active constraints. Detecting passive occurrences is very important, not only because superfluous searches for join partners are avoided, but also because index structures only required for these searches can be discarded, and because they improve the results of static control flow analyses.

We now review a number of standard passive occurrence optimisations. We also generalise the passiveness idea to passive negated occurrences.

Subsumption optimisation One occurrence *subsumes* another occurrence if each constraint that matches the latter also matches the former, taking into account join partners, negation conditions and guards. If an occurrence is subsumed by another occurrence in the same non-propagation rule, the former occurrence can be made passive. If an occurrence is subsumed by occurrences in earlier rules (of higher priority), subsumption analysis may also detect redundant rules (often indications of programming mistakes). This standard CHR optimisation is best described by Sneyers et al. (2008). In JCHR2, we extended this principle to rules that contain the negation-based pattern of Example 8.10.

Example 8.15. Both Example 8.9 and Listing 8.8 contain a rule from the WALTZ program with two **edge** occurrences that subsume each other. In both cases, one of these occurrences can thus be made passive.

Example 8.16. The *idempotence* and *antisymmetry* rules of the classical LEQ program similarly contain subsuming occurrences (see Listing 4.1).

Never removed optimisation If no actively removed constraint can match some negative occurrence, that occurrence can be made passive.¹⁴

Never stored optimisation Certain common idioms in rule-based programming involve constraints that are always removed at some point.

Example 8.17. The RAM program contains a rule:

```
illegal @ -pc(_) => fail.
priority illegal = lowest.
```

If fires only if the simulated RAM program reaches an illegal state (the **fail** statement halts execution in a failed state). Such an unconditional removal at a lower priority occurs frequently, and is similar to e.g. default cases in the pattern matching constructs of functional languages, or in switch statements of modern imperative programming languages.

¹⁴Allowing explicit constraint removals from the host language as in Section 5.1.2 complicates the analysis required for this optimisation.

Example 8.18. In the WALTZ program, each rule involves a `stage` occurrence. When all rules of some stage have fired, rules such as the following move to a next stage (again at some lower priority):

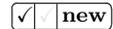
```
done_plotting @ -stage("plot remaining edges") => stage("done").
```

In both cases, constraints are *always removed* at some lower priority. As the following example will help elucidate, this may lead to passive occurrences.

Example 8.19. In the RAM program, all occurrences of the `pc` constraint are removed (all rules resemble the `add` rule of Example 8.1). Combined with the `illegal` rule of Example 8.17, this suggests `pc` constraints should never be stored. If `pc` constraints are made *never stored*, non-`pc` occurrences can never lead to complete rule instances, and can all be made passive. This drastically improves performance: `mem` constraints are never activated, less indexes are built for `prog` constraints, and so on.

For ω_r -based CHR systems the never stored optimisation is relatively straightforward (Duck 2005; Holzbaur et al. 2005). In general, however, for this optimisation to work, constraints must be indexed more eagerly (`mem` and `prog` constraints in Example 8.19) to ensure that the never stored active constraints observe their passive partners. This trade-off with late indexing (Section 8.3.3.1) is not yet fully understood, and should be further investigated.

8.3.4.5 Dynamic passive occurrences



Recently, in JCHR, we added a dynamic variant of the standard never stored optimisation discussed in the previous subsection. If, for some occurrence, a cheap dynamic test shows the constraint store for a required partner constraint is empty, that occurrence, or group of occurrences, is skipped during a constraint's activation. This optimisation manages to compensate for those cases where the compiler fails to derive the never stored property of constraints. It also facilitates the lazy indexing optimisation introduced in Section 8.3.3.4.

Example 8.20. A CHR programmer unaware of the never stored optimisations performed by CHR compilers—and, in fact, we cannot stress enough that programmers should not be expected to learn of such optimisations—is far less likely to include rules such as the RAM program's default rule in Example 8.17. The same is true for many programs. As discussed in Example 8.19, the never stored optimisations are quintessential for the RAM program's performance. Our dynamic never stored optimisation, when combined with lazy indexing, achieves nearly optimal performance even when the default rule is not stated.

Similar ideas have been proposed in production rule literature. Batory (1994) calls it *active rule optimisation*, and Doorenbos (1995) introduced its Rete equivalent, which he coined *unlinking*.

8.3.5 Optimising constraint reactivation

CHR constraints are reactivated when newly added built-in constraints may enable additional rule applications. Up till now, we have mostly ignored this aspect of CHR evaluation. We now survey techniques used to reduce reactivation overhead. While we only discuss built-in constraints here, in most systems these techniques should be extended to arbitrary host statements (see also Chapter 7).

8.3.5.1 Selective reactivation



A naive approach reactivates all (non-fixed, cf. Definition 4.10) stored constraints for each built-in constraint added. This corresponds to the unoptimised **Solve** transition in the refined operational semantics (Section 4.2.3). Always reconsidering *all* constraints though is clearly excessive. Changes in a built-in constraint store are typically restricted to a limited set of variables. Obviously, only those CHR constraints whose arguments contain these affected variables should be reactivated. Essentially, we therefore add to every constrained (logical) variable a list of references to CHR constraints it is involved in. These are used by built-in solvers to selectively reactivate affected constraints only. In OO terms this corresponds to the observer design pattern (Gamma et al. 1995). Analogous techniques are used by typical CP constraint solvers (Rossi et al. 2006). Implementations of CLP and CHR(Prolog) libraries typically use *attributed variables* for this (Holzbaur 1992; Holzbaur and Frühwirth 1999; Schrijvers and Demoen 2004b).

Several additional optimisations are possible to further improve the selectivity of constraint reactivations:

- If two unbound (logical) variables are told equal, only the constraints observing one of these variables have to be reactivated. This is correct because all rules that become applicable by telling this equality constraint necessarily contain constraints over both variables.
- Often, the outcome of an occurrence's guard is known never to change by adding built-in constraints (e.g. if there is no guard, or the constraint's arguments are ground). In this case, there is no need to reconsider this occurrence during reactivation. The property is called *anti-monotonicity*, and is treated in detail in Section 10.2.1.
- Duck et al. (2003) employ so-called *wake conditions* for a more fine grained selection of possibly affected occurrences that are reconsidered after reactivation. This becomes particularly interesting for more complex built-in constraints, such as finite domain constraints.

Refined operational semantics of CHR typically prohibit these last two optimisations that selectively skip occurrences on reactivation. This is an indication that such semantics may be too deterministic (see however the caveat for the generation optimisation in Section 8.3.5.3).

8.3.5.2 Delay avoidance



In the previous subsection, we already briefly introduced anti-monotonous guards, a term coined by Schrijvers and Demoen (2004a) which we discuss in more detail in Section 10.2.1. If all guards on some argument of a particular constraint predicate are anti-monotonous, constraints of that predicate do not have to register as observer of that argument. Schrijvers and Demoen (2004a) called this *delay avoidance*. Technically, this optimisation is again not allowed by the standard refined operational semantics (Section 4.2.3). The formal adjustments required to ω_r 's **Solve** transition are discussed in Appendix B.

8.3.5.3 Generation optimisation



In an ω_r -based system, if a rule fires, the active constraint is suspended until the body is fully evaluated. During this time, this suspended constraint may be reactivated. When the execution continues with the suspended constraint, rules that match that constraint have already been fired during its reactivation. Searching for more partner constraints, and continuing with further occurrences, is then superfluous. This may save a lot of redundant work.

This optimisation has been implemented in several ways. Schrijvers (2005) uses an integer field called the generation of a constraint (incremented each time the constraint is reactivated). In (Van Weert et al. 2008; Van Weert 2008a), we proposed a slightly more efficient version based on boolean field (set to **false** before the body, and to **true** *after* each reactivation). We moreover extended this idea to prevent the same constraint from being reactivated twice.

One caveat though that has never been mentioned: in general, the correctness of the generation optimisation (see Schrijvers 2005 for a formal proof) requires that all occurrences are tried during reactivation. Care must therefore be taken when combining it with the selective reactivation optimisations discussed in Section 8.3.5.1.

In CHR^P and our current CHR² compilation scheme, the standard generation optimisation is not applicable, since only rules of strictly higher priority are considered¹⁵ while an active constraint is suspended. It does give rise to a related problem though, where multiple instances of the same constraint are either active or on the schedule at different priorities. Properly filtering such redundant continuations is part of future work.

8.3.6 Program specialisation

Frühwirth (2005d) add redundant, specialised rules to a CHR program; these rules capture the effect of the original program for a particular goal. In more recent work,

¹⁵Once we abandon this restriction in Chapter 9, the generation optimisation becomes applicable again.

Tacchella et al. (2007) adapt the conventional notion of unfolding to CHR. While both these studies only approach program transformation from a theoretical point of view, such techniques are surely interesting for optimisation. In Section 9.4.2, for instance, we discuss some potential unfolding-based optimisations. We now review other practical applications of program specialisation.

8.3.6.1 Constraint specialisation

Sarna-Starosta and Schrijvers (2008b) proposed source-to-source transformations to specialise constraints and rules based on manifest argument values in occurrences. These specialisations improve the accuracy of static program analyses, constraint indexing, and occurrence dispatch (see also Section 8.6.2).

8.3.6.2 Guard simplification extended

For each occurrence, guard simplification looks at removed occurrences earlier in the active constraint's occurrence order to remove redundant guards, thus also facilitating other analyses such as passive occurrence detection. The most powerful guard reasoner for CHR is created by Sneyers et al. (2008).

We extended the guard simplification principle to negation conditions:

Example 8.21. The following excerpt, adapted from the DIJKSTRA program by Sneyers et al. (2006a), illustrates a common idiom with negation:

```
scanned      @ -relabel(N), +dist(N, _).
not_scanned @ -relabel(N), ~dist(N, _) => ...
```

No matter which occurrence an active `relabel` constraint considers first, if the that rule does not fire, the other rule is always applicable, and the second `dist` lookup can be omitted. It also becomes apparent that the `relabel` constraint is never stored, which may enable many additional optimisations (see Section 8.3.4.4).

Example 8.22. If a constraint matching a negated condition is always removed at some higher priority, the tests for these negated conditions can be omitted.

8.4 Evaluation

8.4.1 CHR systems

In this section, we compare the performance of JCHR with other CHR systems. An overview of these systems was given earlier in Section 4.5.3.

In (Van Weert et al. 2005), we compared JCHR with two other CHR(Java) systems, JaCK and DJCHR. The conclusion was clear: due to its optimising

	JCHR	YAP	SWI	SICStus	CCHR
LEQ(100)	0.09	5.81 (1.55%)	13.4 (0.67%)	9.64 (0.93%)	0.29 (30.9%)
PRIMES(4096)	0.27	2.68 (10.2%)	4.79 (5.69%)	4.15 (6.57%)	0.25 (107%)
TAK(500,450,405)	0.05	0.19 (26.0%)	0.54 (9.25%)	0.86 (5.84%)	0.07 (68.8%)
DIJKSTRA(16384)	1.96	3.59 (54.6%)	9.23 (21.3%)	34.2 (5.73%)	1.98 (98.8%)
RAM_FIB(200k)	0.77	25.1 (3.07%)	<i>stack overflow</i>	95.1 (0.81%)	5.47 (14.1%)
UNION(50k)	0.41	1.38 (29.3%)	3.39 (12.0%)	8.70 (4.67%)	294 (138%)

Table 8.1: Benchmark comparing performance for some typical CHR programs in several systems. The average runtime in seconds is given with between parentheses the relative performance of JCHR.

compilation scheme and efficient indexing, JCHR’s performance was far superior both in time and in space, typically by many orders of magnitude, and often with a better complexity. As JCHR is the only CHR(Java) system that has evolved since, we do not repeat these results here.

For this section, we compared JCHR with more recent, state-of-the-art systems: the K.U.Leuven CHR system for YAP, SWI, and SICStus Prolog, and the CCHR system for C. Table 8.1 shows the results. The imperative CHR systems JCHR and CCHR are significantly faster than the Prolog systems, typically by one or two orders of magnitude. This is partly because the generated Java and C code is (just-in-time) compiled, whereas the Prolog code is interpreted. Other contributing factors are the more efficient indexing structures, and different analyses and optimisations that are performed: Table 8.3 provides an overview. The performance of JCHR and CCHR is mostly about the same.

8.4.2 Production rule systems

We compared JCHR2 with two established production rule engines. Clips was chosen as a reference, because a recent performance survey suggests it is currently the most efficient system available (IllationTM 2007). Jess is a more recent Java-based implementation. We refer to Section 2.2.2 for a brief discussion on these systems. The results measured are shown in Table 8.2.

We used standard CHR and production rule benchmarks only, without optimising the programs for either system. The three systems ran equivalent programs (using automatic source-to-source transformation mostly), and fired exactly the same rules. Only for the nondeterministic WaltzDB benchmark, the three systems fired slightly different rule instances.

The Clips-to-JCHR transformation used for the last six benchmarks declares all constraints as `set` constraints (see Section 5.1.6). By manually adding better invariant declarations—such as functional dependencies and unenforced invariants (cf. Section 8.3.1.2)—further performance gains were possible: MANNERS(256) for example ran in 70ms, WALTZ(100) in 617ms, about 3 and 2 times faster resp.

	CLIPS 6.3		Jess 7.1		JCHR2	
DIJKSTRA(8192)	16.3	34	22.2 (136%)	25	1.24 (7.60%)	447
PRIMES(2048)	2.02	1.88	3.59 (178%)	1.05	0.07 (3.46%)	54.2
RAM_FIB(50k)	12.2	45.1	36.1 (296%)	15.2	0.69 (5.62%)	803
UNION(25k)	40.7	5.5	10.4 (25.6%)	21.6	0.28 (0.70%)	793
MANNERS(256)	48.6	0.69	361 (742%)	0.09	0.19 (0.39%)	202
SUDOKU 3x3-p17	0.85	9.59	4.57 (539%)	1.78	0.44 (52.3%)	19.8
3x3-p17 (stress)	3.06	3.2	54 (1754%)	0.18	1.10 (36%)	8.96
WALTZ(100)	6.25	4.47	32.1 (513%)	0.87	1.12 (18%)	24.9
WALTZDB(32)	2.8	9.52	12.9 (461%)	2.01	0.42 (15.1%)	53.7
WORDGAME(200)	4.49	4.55	5.51 (123%)	3.72	2.92 (65.1%)	6.98

Table 8.2: Comparison with state-of-the-art production rule systems. For each system, a first column gives the average running time in seconds (between parentheses is the relative performance compared to Clips), and a second the average rule application rate in kRAPs (1,000 *Rule Applications Per Second*).

This clearly shows the importance of invariant annotations.

JCHR2 outperformed state-of-the-art production rule engines, often by several orders of magnitude. Admittedly, this comparison should be taken with a grain of salt, as JCHR2 is the only system that compiles its programs. However, since Clips is implemented in C rather than Java, one could argue that the comparison between JCHR and Clips is reasonably fair.

More importantly though: JCHR mostly scales better with larger problem sizes, often with better asymptotic time complexity. This is clearly shown in Figure 8.1. These results are in line with earlier findings that compare lazy matching algorithms similar to ours with Rete (Miranker et al. 1991; Obermeyer and Miranker 1994). We discuss this related work shortly in Section 8.5.

Finally, while measuring and comparing actual memory usage of different systems is hard, our experiments did confirm space performance of lazy evaluation is superior as well. Space complexity is also discussed in Section 8.5.

8.5 Discussion and Related Work

8.5.1 CHR systems

An overview of existing CHR systems and their basic implementation methodology was given earlier in Section 4.5. In Table 8.3, we provide a brief overview of the optimisations implemented by the K.U.Leuven CHR, JCHR and CCHR systems. The extent in which other systems apply these optimisations varies greatly. HALCHR, for instance, is a similarly optimising compiler for HAL, whereas most CHR(FP) and CHR(Java) systems are relatively naive interpreter-based

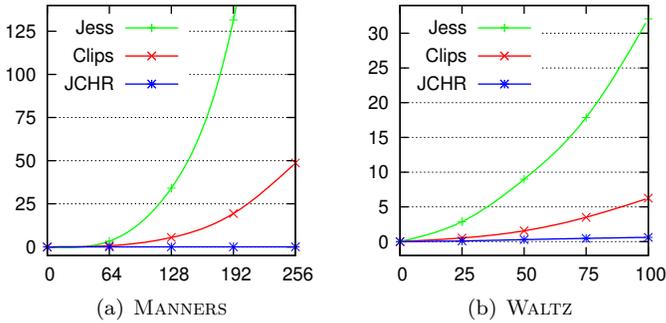


Figure 8.1: Performance comparison for two famous benchmarks. Graphs show average total runtime in seconds.

prototypes. We refer to Section 4.5 for additional references.

8.5.2 Production rule systems

In Section 2.2.3, we briefly introduced the most important matching algorithms used by production rule systems. Our basic lazy evaluation scheme of Section 8.2.3 for CHR with negation is very similar to the LEAPS algorithm by Miranker et al. (1990). As discussed in detail in Section 10.5, the main difference is the way reapplication is prevented. Even the most elaborate description of LEAPS though, that by Brant (1993), does not describe how to incorporate priorities, or how to optimise beyond the basic algorithm of Section 8.2.3.¹⁶ The only publication on LEAPS optimisation know of is (Obermeyer et al. 1995), where the importance of indexing is stressed.

On possible optimisations and variants of Rete-like matching algorithms, on the other hand, there is a considerable literature. We cannot possibly cover all related work, and refer to (Doorenbos 1995) for an excellent introduction of Rete and its most common optimisations. For specific optimisations, we refer to the related work sections of the many CHR publications cited earlier. To the best of our knowledge though, many of the optimisations and analyses in this chapter have never been applied to production rule systems.

In Section 2.2.3, we already discussed in sufficient detail why Rete’s performance is often lacking. The two fundamental reasons are the eager computation of all applicable rule instances, and the injudicious application of join indexing (beta memories). This is well-documented in production rule literature, and several alternative matching algorithms have since focused on improving join indexing

¹⁶More precisely, LEAPS corresponds to a naive version of the CHR compilation scheme presented later in Chapter 9; we discuss this in Section 9.4.1.

<i>Optimisation</i>	<i>Prolog</i>	<i>JCHR</i>	<i>CCHR</i>
Invariant derivation (§8.3.1.1)	✓	✓	
Unenforced invariants (§8.3.1.2)			
Loop-invariant code motion (§8.3.2.1)	✓	✓	✓
Indexing (§8.3.2.2)	✓	✓	✓
Exploiting invariants (§8.3.2.3)		✓	
Pre-commit backjumping (§8.3.2.4)		✓	
Post-commit backjumping (§8.3.2.5)		✓	✓
Fragile iterators (§8.3.2.6)	±	✓	✓
Join ordering (§8.3.2.7)	±	✓	±
Late storage (§8.3.3.1)	✓	✓	±
Late allocation (§8.3.3.2)	✓		±
Memory reuse (§8.3.3.3)			±
Lazy indexing (§8.3.3.4)		✓	
Passive occurrences (§8.3.4.4)	✓	✓	±
Dynamic passive occurrences (§8.3.4.5)		✓	
Selective constraint reactivation (§8.3.5.1)	✓	✓	✓
Delay avoidance (§8.3.5.2)	✓	✓	
Generations (§8.3.5.3)	✓	✓	✓
Constraint specialisation (§8.3.6.1)			
Guard simplification (§8.3.6.2)	✓		
Recursion optimization (§9.2)		✓	✓
Distributed history (§10.1)	✓	✓	✓
Non-reactive history elimination (§10.2)	✓	✓	
Idempotence (§10.3)	✓	✓	

Table 8.3: Summary of all relevant ω_r -related optimisations and their implementations in K.U.Leuven CHR, K.U.Leuven JCHR and CCHR (development versions at time of writing). Of course, this overview does not fully reflect the considerable difference in strength of the static analyses, the extent of optimisation, and runtime data structures. Optimisations though that are clearly only implemented in an ad-hoc or restricted manner are indicated with ‘±’.

performance. We refer to Section 2.2.3 for details and references.

For improved performance, several recent rule engines provide a so-called *sequential* matching algorithm. From a given working memory, this algorithm first computes the conflict set, and then fires all computed instances, but *without activating facts* (i.e., without inserting assertions and retractions in the Rete network). This seems a rather ad hoc workaround to the inherent overhead of the Rete network. Clearly, lazy matching is a much cleaner, more satisfactory solution to Rete's performance issues.

Worst-case complexities

In our experience, lazy matching is often poorly understood. It has barely received any attention by production rule research, and has the reputation of being complex and difficult to comprehend (Batory et al. 1994). A common belief is that LEAPS has better asymptotic space and time complexity than Rete and TREAT. While in practice this is mostly true, general worst-case claims such as in (Batory et al. 1994; Miranker 1998) typically do not hold.

Time complexity Firstly, time complexity is often in the first place dominated by orthogonal factors, such as join ordering and proper indexing. But even when ignoring this, it should be clear that for time complexity no general claims can hold when comparing LEAPS, Rete and TREAT. An example is trivially constructed, for instance, where join indexing is required for the optimal time complexity. And conversely, it is equally straightforward to see that eagerly computing all applicable rule instances may lead to worse time complexity. We do believe though that a lazy matching algorithm, when combined with proper *selective* join indexing (cf. Section 8.6), should in principle have superior time complexity in practice.

Space complexity In Chapter 10, we show that the worst-case space complexity of CHR's lazy evaluation strategy is $\mathcal{O}(|\mathbb{S}|^n)$, with n is the maximum number of (positive) occurrences in a rule's head. This corresponds to the size of the propagation history.¹⁷ Brant (1993) determined the same holds for LEAPS.¹⁸ Clearly, the worst-case space complexity of eager matching algorithms is also at least $\mathcal{O}(|\mathbb{S}|^n)$, the size of the conflict set. TREAT, for instance, has this exact same space complexity. The beta network that Rete keeps, however, may indeed consume more space in the worst case. Brant (1993) uses the overly conservative bound of $\mathcal{O}(|\mathbb{S}|^{n+m})$, with $n+m$ the largest total number of occurrences in a rule, counting both positive and negative occurrences. For typical implementations of

¹⁷Provided the history is optimally garbage collected (Section 10.1). The worst-case is attained if this n -headed rule is reactive (Section 10.2), or contains negation (Section 10.5.2).

¹⁸At least for OPS5 rules; cf. Section 10.5.2.

Rete though, tighter bounds can be derived, only slightly worse than that of lazy matching. For the OPS5 rules that Brant (1993) considers, for instance, Rete's worst-case space complexity is actually $\mathcal{O}(|\mathcal{S}|^{n+1})$, or even just $\mathcal{O}(|\mathcal{S}|^n)$ if a more efficient implementation of negated conditions is used. The actual complexity depends on the way negative conditions are implemented, and the extent to which join indexing is used.

Still, in practice, the actual space behaviour of lazy matching algorithms is indeed typically superior. In Chapter 10, we determine that most rules and programs do not require a history. For positive-only, non-reactive programs, for instance, the worst-case space complexity is only $\mathcal{O}(|\mathcal{S}|)$ (Brant (1993) showed this also holds for LEAPS). For CHR and LEAPS, the worst-case space consumption is thus hardly ever reached in practice. The actual memory consumption measured for algorithms that perform eager matching or join indexing though, rapidly reaches complexities close to their worst case bounds.

8.6 Ongoing and Future Work

Despite the already generally positive results, there is still considerable room for improvement. By now, the behaviour and optimisation of ω_r -based systems is by quite well understood. The deterministic nature of ω_r though made this still relatively easy, when compared to e.g. the highly nondeterministic ω_2 semantics. With JCHR2, we only have just begun researching how to optimally exploit the added freedom this semantics offers. While more challenging, this freedom should be seen as an important opportunity for more advanced, automated optimisation of CHR programs.

Some other, more specific important areas of future work include:

Join indexing Join indexing is a technique extensively used by Rete-like matching algorithms, as introduced in Section 2.2.3. As far as we know, lazy evaluation has never been combined with join indexing. In certain cases, a judicious application of join indexing, however, is required for an optimal runtime complexity. Static or dynamic techniques have to be investigated to determine when to use join indexing: see also Section 8.6.1.

Concurrency Leveraging the full power of current and future multi-core and many-core processors demands highly concurrent software. In fact, in Section 11.2, we consider this one of the ‘grand challenges’ of CHR. Many important problems are still open in this area, from language features and semantics, to analysis, implementation, and optimisation.

In the next two subsections we discuss some promising research directions we are currently pursuing in a bit more detail.

8.6.1 Dynamic optimisations

CHR research and implementation so far has mainly focused on the static analysis and optimisation of programs. Static techniques often fail though. In these cases, dynamic optimisations should be investigated.

We already added some initial dynamic optimisations to JCHR2, such as lazy indexing (Section 8.3.3.4) and dynamic passive occurrences (Section 8.3.4.5). While results are promising, these contributions only constitute a first, relatively small step towards dynamically optimising rule evaluation. Dynamic counterparts for indexing and join ordering in particular are urgently needed. As an initial, exploratory approach, we are also considering program recompilation using runtime statistics gathered by profiling tools.

Dynamic indexing

As discussed earlier, both fact and join indexing inherently involve non-negligible maintenance overhead. Statically deciding which indexes to use is hard, and the eager approach taken by current CHR compilers (and current production rule systems as well) frequently leads to index data structures with a disproportionate overhead. Possible dynamic optimisation techniques include the dynamic adding or removing of indexes, or on-demand index population. The results obtained with our still relatively simple lazy indexing strategy show such optimisations are crucial. In the context of eager matching algorithms, related ideas are proposed e.g. by Fabret et al. (1993), Hanson et al. (1995), and Wright and Marshall (2003) (cf. also Section 2.2.3).

Dynamic join ordering

Unlike Miranker and Lofaso (1991), we strongly believe static join ordering alone is insufficient. Functional dependencies and the cardinality annotations proposed in (Van Weert et al. 2009) help, but in general the lack of information renders it impossible to statically finding the optimal join ordering. In the worst case, there even is no single optimal join ordering, when the right join order depends on runtime characteristics (cf. De Koninck and Sneyers 2007). A wrong join order often results in suboptimal time complexity.

Unfortunately, dynamic join ordering does not match well with the static compilation techniques used by current state-of-the-art CHR systems. Ideally, a more dynamic interpreter-based execution should be used, preferably combined with just-in-time compilation. In other words, this would require us to completely redesign our CHR systems. Such a redesign is arguably also already required though to improve the scalability of CHR systems to larger, real-life programs (one of the ‘grand challenges’ listed in Section 11.2).

8.6.2 Global optimisations

CHR compilers currently focus on *locally* optimising the join computation of individual occurrences. For larger programs, global optimisations are in order:

Optimised occurrence dispatch

By default, an active constraint linearly traverses all occurrences, even when only a small subset may lead to rule instances.

Example 8.23. A typical example is the **stage** constraint in the WALTZ program (see Examples 8.9 and 8.18). Depending on its single argument, only a very restricted subset of the occurrences should be considered.

For this example, using switch statements significantly improves performance. In general, the **activate** procedure of Fig. 8.4 should where possible incorporate guards or negated conditions to improve occurrence dispatch. This is clearly an area where we should profit from Rete and TREAT research (these algorithms perform occurrence dispatch using a so-called *alpha network*).

Inlining and merging occurrence procedures

Several occurrences (of the same priority) often compute similar joins. Their procedures can be inlined and merged to produce more efficient code.

Example 8.24. Most rules in the RAM program have the same form as the **add** rule used in our running example. For each occurrence, an active **pc** fact first retrieves a **prog** fact. This lookup should be done only once. Next, a switch statement on its instruction field could single out which rule is applicable in constant time.

Clearly, such optimisations go well beyond simple occurrence dispatch. They also subsume optimisations such as guard simplification, and facilitate several others as well. We believe them to be an important next step in the state-of-the-art of efficient rule execution.

Such optimisations require a considerable effort. The current, preliminary implementation of this optimisation in JCHR2 shows mixed results. For the RAM program, for instance, it improves performance considerably. For several other programs, however, performance is reduced, for different reasons. Additional research is required to determine when to use occurrence merging.

Chapter 9

Recursion Optimisations

The proverbial German phenomenon of the verb-at-the-end about which droll tales of absentminded professors who would begin a sentence, ramble on for an entire lecture, and then finish up by rattling off a string of verbs by which their audience, for whom the stack had long since lost its coherence, would be totally nonplussed, are told, is an excellent example of linguistic recursion.

— **Douglas Hofstadter** (born 1945)

American mathematician, cognitive scientist, and author

In this chapter, we identify several challenges related to the evaluation of recursive CHR programs. We show and explain why these issues are particularly grave when compiling to imperative target languages. We therefore developed a new optimised compilation scheme and runtime system, capable of efficiently executing sequential conjunction and recursion.

While this research was initially performed in the context of ω_r -based systems (Van Weert, Wuille, Schrijvers, and Demoen 2008), the same technical issues resurface when adding sequential conjunctions to the compilation scheme for CHR². Chapter 8 only considered rule bodies with batch semantics. This allowed us to sidestep recursion issues by only allowing a single active constraint per priority level (cf. Section 8.2.4). In general, the body of a normalised CHR² rule is a sequential conjunction of batch conjunctions (cf. Sections 5.2.1 or 8.1). For such rules in particular, recursion must be treated differently.

Section 9.1.1 starts by outlining the obvious, naive way of adding sequential conjunctions to CHR²'s compilation scheme, using an approach similar to the one taken by traditional ω_r -based systems. In Section 9.1.2 then, we show this leads to critical performance issues with the host's (implicit) call stack. Our

analysis explains why imperative target languages are particularly ill-equipped to deal with the naive code generated for recursive CHR programs. With this in mind, we completely redesigned the execution methodology of our JCHR and CCHR systems (Van Weert et al. 2008; Van Weert 2008a). In JCHR2, we have since extended these techniques to deal with CHR2’s batch conjunctions and rule priorities. We discuss these contributions in Section 9.2.

9.1 Sequential Conjunctions and Recursion

9.1.1 Basic compilation scheme

Naively adding sequential conjunction to the basic compilation scheme of Section 8.2.2, i.e. the one without priorities, is quite straightforward. We simply repeat lines 14–18 in Listing 8.1 once for each batch conjunction in the sequence, thus adding and activating the constraints batch per batch. An important point to note is that for purely sequential conjunctions, this reduces precisely to the standard scheme used by ω_r -based systems (Holzbaur and Frühwirth 2000a; Duck 2005; Schrijvers 2005; Van Weert et al. 2008).

Effectively combining sequential conjunction with the schedule-based solution for priorities of Section 8.2.4 requires two additional changes though. First, recall the semantics of sequential conjunctions established in Chapter 5. In essence, it requires that before moving to a next sequential conjunct, *at least* all those rule instances have fired that both:

1. have higher *or equal* priority; and
2. *became* applicable by, or at some point after, adding the constraints of the previous sequential conjunct

Any natural, meaningful semantics for sequential conjunctions must have these same properties. We now discuss both above items in turn.

Firstly, recall from Section 8.2.4 that our initial scheme does not consider occurrences with a priority *equal* to that of the previously fired rule; i.e. it only considers occurrences of *strictly higher* priority. De Koninck (2008) explicitly preferred this unusual execution strategy because it eases certain program analyses and optimisations. Even for pure batch semantics though, we already find this often results in unexpected behaviour. But regardless, for sequential conjunctions, it clearly is no longer an option. As a first step, we therefore adjust the comparisons with p_{active} to include equal priorities as well, both on line 9 of Listing 8.4, and in Section 8.2.4’s `activate` procedure.

Secondly, when a rule is fired at some priority, the schedule often already contains continuations of equal priority. By the above revision of the `activate` procedure, these now become activated as well. In other words: not only those rule instances that became applicable by evaluating a sequential conjunct are

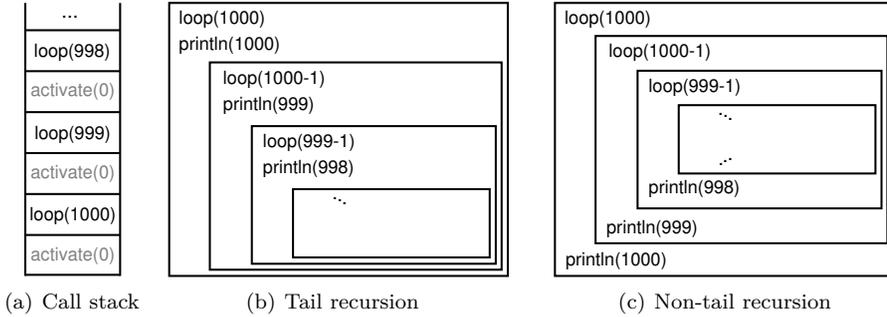


Figure 9.1: Execution of recursive CHR rules.

fired, but possibly also some that were already applicable before. While this is allowed by the ω_2 semantics, it arguably leads to unexpected runtime behaviour. We therefore introduced two new schedule operations:

`saveSchedule(p_{active})` All items that are scheduled at p_{active} when this operation is called are ignored by subsequent calls of `scheduledPriority` and `pollScheduled`; items scheduled later are still observed as normally.

`restoreSchedule()` Restores the schedule items saved by the last call of `saveSchedule(p_{active})` (only called if `scheduledPriority()` > p_{active}).

Calls to these operations are added before and after a sequential rule body.

9.1.2 Problem analysis: recursion and stack overflows

As most declarative programming languages, CHR does not provide language primitives for loops. Any non-trivial CHR program therefore contains recursion. For such programs, the naive compilation scheme of Section 9.1.1 generates a set of mutually recursive host language procedures. It relies on the host language’s compiler or interpreter to adequately deal with recursion. If not, the naive compilation scheme will typically lead to *call stack overflows*.

Example 9.1. To demonstrate this elementary issue, we use this CHR rule:

$$\text{tail @ loop}(N) \Leftrightarrow N > 0 \mid \text{println}(N), \text{loop}(N-1).$$

and its variant obtained by reversing the order of the sequential conjunction:

$$\text{non-tail @ loop}(N) \Leftrightarrow N > 0 \mid \text{loop}(N-1), \text{println}(N).$$

The execution of the query `loop(1000)` using these rules is illustrated in Figures 9.1(b) and 9.1(c) respectively. Note that by reversing the order of the conjunction, the order in which the numbers are printed is reversed as well. So both programs are by no means equivalent.

Figure 9.1(a) shows the resulting call stack when using naive call-stack-based execution. In ω_r -based systems, the interleaving calls of `activate(0)` that result from our unoptimised¹ schedule-based scheme are not present. Without optimisation, the call stack thus always consumes space linear in the number of recursive calls. A *stack overflow* occurs when the space allocated for the call stack is insufficient to hold the required stack frames.

The empirical results of Section 9.1.3 confirm that for recursive CHR programs stack overflows are indeed problematic. But what they mostly show is that for imperative CHR systems, the problem is significantly worse. In the next subsection, we briefly explain this observation.

Tail call optimisations

One technique used to reduce the performance impact of recursion is called *tail call optimisation*. Particularly for programming languages that advocate a recursive programming style, such as most logical and functional languages, tail call optimisation is considered indispensable. A procedure call is said to be in tail position if it is the last operation executed before the calling procedure returns. In Example 9.1, for instance, the call of `loop(N-1)` in the `tail` rule is a tail call. As clearly visible in Figure 9.1(b), when execution returns from a tail call, no further operations are required, and execution immediately returns to the previous caller. The idea of tail call optimisation is that, instead of adding a new stack frame for each tail call, the stack frame of the previous caller is simply overwritten, such that control immediately returns there instead.

Even though similar optimisations are in principle possible in imperative host languages (Probst 2001), in practice, they are only scarcely performed. The GCC C compiler (Free Software Foundation 2010), for instance, only optimises tail calls in specific cases (Bauer 2003). Typical implementations of the Java Virtual Machine (Lindholm and Yellin 1999), including the reference implementation HotSpot (Sun Microsystems, Inc. 2010), do not perform tail call optimisations at all.²

Tail calls in CHR

For CHR rules, intuitions regarding tail calls can be deceiving. That is: the last conjunct of a rule's body does not necessarily correspond to a tail call. In most cases, if the active constraint matches a kept occurrence, more partner constraints must still be searched, or more occurrences tried, after the evaluation

¹For presentation purposes, we ignore for now the schedule-related optimisations discussed earlier in Section 8.3.4.2. Their impact is discussed in Section 9.2.1.

²One often cited reason is that tail call optimisations would interfere with Java's stack walking security mechanism (though this security folklore has recently been challenged by Clements and Felleisen 2004).

	JCHR		CCHR	SWI	YAP
	JRE 1.5	JRE 1.6			
tail	35,900	3,200	∞	∞	∞
non-tail	38,700	3,200	0.5M	3.3M	$\pm\infty$

Table 9.1: Recursion limits for different CHR systems. The number indicate the approximate value of N for which the recursive rules of Example 9.1 resulted in stack overflow for the different systems when called with initial query `loop(N)`. Here, ∞ indicates the program ran in constant space, and $\pm\infty$ indicates the limit was available (virtual) memory.

of the last body conjunct. For simpagation rules, therefore, the applicability of tail call optimisations may even depend on which constraint is active.

It is therefore much less likely that recursive CHR programs can be rewritten to use tail recursion than in e.g. in typical functional or logical programming languages. This means that even for CHR(Prolog) systems, many recursive CHR programs will put a heavy burden on the call stack. However, as shown next by the numbers in Section 9.1.3, unlike imperative execution environments, Prolog interpreters are typically designed to withstand deeply recursive clauses.

9.1.3 Problem demonstration: empirical results

We used the rules of Example 9.1 to test the approximate recursion limits for different CHR implementations. Table 9.1 contains the results measured for the original K.U.Leuven JCHR and CCHR systems, compared with those of the K.U.Leuven CHR implementations in SWI-Prolog and YAP Prolog.

For all systems but JCHR, the `tail` rule ran in constant space. Note that this is an example where even the GCC compiler performs the required tail call. This is not always the case though (Bauer 2003). Executing the compiled JCHR handler with the HotSpot Client JVM (Sun Microsystems, Inc. 2010), rapidly resulted in stack overflow. Using version 1.5 of the Java Runtime Environment (JRE), a stack overflow occurred for N equal to 35,900. With JRE 1.6 the situation even worsened by another order of magnitude.

For the rule without tail recursion, the results for JCHR of course remained unchanged. For both SWI Prolog and CCHR, the native call stack has a static upper bound. For CCHR, the test resulted in stack overflow after around half a million recursive calls, for SWI after around 3.3 million. YAP Prolog’s call stack grows dynamically, so YAP is only limited by available (virtual) memory. These numbers demonstrate that Prolog systems shine at handling deep recursive calls, even if tail call optimisation is not applicable. Programs written in imperative languages, on the other hand, are clearly expected to use iteration instead of recursion.

	JCHR	SWI		JCHR	SWI	
BEER(N)	3,500	∞		PRIMES(N)	4,800	∞
DIJKSTRA(N)	2,100	∞		PRIMES_SWAPPED(N)	4,800	3.4M
FIBBO(N)	1,800	<i>timeout</i>		RAM_FIB(N)	300	20,000
GCD(N)	4,300	4.5M				

Table 9.2: Recursion limits for several standard CHR benchmark programs (JRE 1.6 was used for JCHR). The numbers in the second and third column indicate the approximate value of N for which the benchmark results in stack overflow; ∞ indicates the benchmark ran in constant stack space.

We also tested the recursion limits for more realistic CHR benchmark programs. Table 9.2 confirms that the performance of CHR’s traditional compilation scheme is unacceptable in Java. JCHR’s poor performance on the DIJKSTRA problem was also observed by Sneyers et al. (2006a, 2009). Due to the lack of tail call optimisations and the limited size of the call stack, stack overflows occur unacceptably fast when executing recursive JCHR programs. Depending on the version and platform, this can already be after a few thousand recursive calls.

Obvious examples of CHR programs that should, but do not, run in constant stack space include PRIMES and RAM. For the latter, SWI Prolog also does not manage to perform tail call optimisation. To guarantee executions with optimal space complexity of such CHR programs (cf. also Sneyers et al. 2009’s seminal result discussed in Section 4.3.3 which relies on this RAM program), CHR compilers should therefore perform tail call optimisations themselves.

9.2 Recursion Optimisations

Since improving the optimisations of the host environment is seldom an option, we designed novel compilation schemes for CHR that avoid execution stack overflows. The first is very easy to implement, but deals only with tail recursion (Section 9.2.1). The second typically takes a substantial implementation effort, but is necessary for the efficient execution of more general recursive CHR programs in imperative host languages (Section 9.2.2).

9.2.1 Trampoline-based execution

A first optimisation we worked out is based on a popular technique to eliminate tail calls, called a ‘*trampoline*’ (Baker 1995; Ganz, Friedman, and Wand 1999). The basic idea is that, by default, tail calls are no longer called directly. Instead, a *closure* representing this tail call is returned to a loop that is running lower on the call stack.

We described such a trampoline-based scheme for ω_r -based (imperative) CHR systems in detail in (Van Weert, Wuille, Schrijvers, and Demoen 2008). In essence, the execution scheme looks as follows, with \mathcal{C} a CHR constraint:

```

procedure trampoline( $\mathcal{C}$ )
  while  $\mathcal{C} \neq \text{nil}$ 
     $\mathcal{C} = \text{activate}(\mathcal{C})$ 
  end
end

```

As long as only tail recursion is used, the call stack never grows much higher than this trampoline loop, and tail-recursive CHR programs run in optimal constant stack space.

For our approach for general CHR2 programs of Section 9.1.1, it turns out that we get trampoline-like behaviour almost for free. Concretely, the schedule-related optimisation ideas of Section 8.3.4.2 simply have to be applied to tail calls. In a naive implementation, the call stack for tail recursive rules would look like the one in Figure 9.1(a). Note that the `activate(p)` calls interleaving the actual rule applications even preclude tail call optimisations by the host. In Section 8.3.4.2 though, we already suggested that if the active constraint is removed, no new `activate` loop should be started. By applying the same reasoning on tail calls, the `activate` loops effectively act as trampolines, and tail calls no longer consume stack space.

9.2.2 Explicit stack

Trampoline-style execution only works for tail calls. Many (mutually) recursive CHR rules though are not tail recursive. Often, these cannot even easily be rewritten to such a form (Section 9.1.2). Such programs thus require large call stacks to execute, which in imperative languages, and in Java in particular, very easily results in stack overflows (Section 9.1.3). We therefore redesigned JCHR's and CCHR's execution strategies to explicitly manage data structures for the control flow of a CHR program themselves. It might seem that the overflow is just shifted from the stack to the heap. However, for a language like Java, the heap is typically substantially larger than the call stack.³ Moreover, we of course still make sure tail calls run in constant space.

Our high-level presentation here highlights the basic principles of JCHR2's execution strategy. It extends the techniques we introduced in (Van Weert et al. 2008; Van Weert 2008a) towards batch conjunctions and priorities.

³If the standard heap size is insufficient, the HotSpot JRE can be configured to allocate more heap memory. A similar option for the call stack seems to have no effect for all HotSpot versions we tested.

Step 1: continuations To explicitly manage the control flow of a CHR program, we need a mechanism to:

1. save the execution state after each sequential conjunct; and
2. restore this state again later to resume execution from that point.

The general concept that accomplishes this is called a *continuation*, an abstract representation of the control state. In our case, each continuation must represent at least:

- which part of the body to execute next (if any)
- any (still required) local variables
- for kept occurrences, the current active constraint, partner constraints and constraint iterators

While in C implementing continuations is facilitated by `goto` statements, resuming complex nested loops in Java is quite involved (cf. Van Weert 2008a). More detailed information on how to implement continuations for CHR programs is provided in (Van Weert et al. 2008; Van Weert 2008a).

Step 2: execution stack Next, the CHR runtime is extended with a final data structure: the (*CHR*) *execution stack*. Unlike in (Van Weert et al. 2008; Van Weert 2008a), each continuation on the stack also has a priority number associated with it. Obviously, each new continuation that is pushed on the stack will have a priority number less than or equal to all other priorities on the stack. We call the priority associated with the top of the stack, clearly the smallest number (highest priority), the *priority bound*. For an empty stack, this bound is defined as $+\infty$. At runtime, occurrences with a priority number larger than this bound (i.e. occurrences of lower priority) are never considered.

In the basic compilation scheme, once all constraints of a sequential conjunct are scheduled, a continuation is created and pushed onto the stack, and then execution returns to a main control loop. This loop has the following form:

```

1  while true
2      while not scheduleEmpty() and scheduledPrio() ≤ prioBound()
3          activate(pollScheduled())
4      end
5      if stackEmpty() return
6      call(pop())
7  end

```

Scheduled constraints are repeatedly activated until the priority bound is reached (lines 2–4). The constraint activation code of Listing 8.4 (called on line 3) is similarly adjusted to use the priority bound. When line 5 is reached, all newly applicable rule instances of higher or equal priority than the last continuation

pushed on the stack have fired. Calling this continuation (line 6), either executes the next conjunct of a sequential conjunction, resumes the search for more partner constraints, or continues with remaining occurrences of a previously active constraint.

This scheme is a correct implementation of ω_2 . Essentially, ω_2 's stack \mathbb{A} is explicitly managed by the CHR runtime, instead of mapped onto the implicit call stack of the host environment. Moreover, tail calls are executed in constant stack space, as the new loop still acts as a trampoline.

Step 3: optimisation Explicitly maintaining a stack unavoidably entails constant time overheads when compared to the traditional, call-based compilation scheme. The host environment's call stack is able to use more specialised low level mechanisms. This is particularly the case for high-level host languages such as Java. The following minor optimisations are therefore all the more required to reduce this overhead:

- For tail calls, of course no continuation is pushed.
- For certain common patterns, continuations can be reused. JCHR e.g. uses a very efficient `undoPop()` operation instead of creating and pushing a new, equivalent continuation.
- Doubly reactivated constraints are removed from the stack without traversing their occurrences (this an extension of the generation optimisation of Section 8.3.5.3).
- If static control flow analysis shows that activating a constraint (of equal priority) does not result in recursion, the original scheme without the explicit call stack is used.

We refer to (Van Weert 2008a) for details on these optimisations.

9.3 Evaluation

Table 9.3 contains the results measured when comparing JCHR's improved compilation scheme with its traditional one. Our improved scheme solves all stack overflow issues. All tail recursive programs run in constant stack space, and the remaining recursive handlers become limited only by available heap space. From the table it is clear that this more than sufficient.

The numbers also confirm that the optimisations summarised at the end of the previous section manage to reduce the overhead resulting from explicitly maintaining the call stack. There where a comparison is possible, the optimised improved compilation scheme is never more than 20% slower than the traditional scheme. For one benchmark, the improved scheme is even slightly faster.

	Traditional scheme	Improved scheme	
		<i>unoptimised</i>	<i>optimised</i>
BEER(2,000)	7,553	7,341 (-3%)	7,058 (-7%/-4%)
BOOL(1,000,000)	4,106	5,216 (+27%)	4,914 (+20%/-6%)
DIJKSTRA(5,000)	<i>overflow @ 2,100</i>	1,230	1,065 (-13%)
FIB(33)	9,366	12,816 (+37%)	10,934 (+17%/-15%)
GCD(64,000,000)	<i>overflow @ 4,300</i>	5,529	5,519 (-0%)
LEQ(300)	3,821	5,537 (+45%)	4,309 (+13%/-22%)
MERGESORT(100,000)	10,581	12,262 (+16%)	11,310 (+7%/-8%)
PRIMES(10,000)	<i>overflow @ 4,700</i>	4,991	4,396 (-13%)
RAM_FIB(150,000)	<i>overflow @ 300</i>	3,144	2,991 (-5%)
UNION(210,000)	2,637	2,685 (+2%)	2,637 (+0%/-2%)

Table 9.3: Empirical comparison between the different compilation schemes of the K.U.Leuven JCHR system for several recursive CHR programs. The second column gives timings (in average milliseconds) when using the traditional scheme. For the remaining columns, JCHR maintained its own stack. Two versions were tested: a fairly naive one, and one where the optimisations briefly listed at the end of Section 9.2.2 were applied. The percentages between parentheses give the relative difference with the traditional scheme (if applicable), and in the last column also between the unoptimised and the optimised version.

We already verified the competitiveness of our imperative systems using several typical recursive CHR programs in Section 8.4.1. For these benchmarks, JCHR and CCHR efficiently managed their own call stack, performing trampoline-like tail call optimisations there were applicable. The stack overflow reported in Table 8.1 for the RAM program is caused by the SWI Prolog system failing to perform the necessary tail call optimisations (see also Table 9.1).

The benchmark results thus show that, for compiling CHR to imperative languages, and in particular to Java, our new, improved compilation scheme is superior to the traditional one. All stack overflow issues are resolved, and our optimisations reduce the overhead to an acceptable level.

9.4 Conclusions

When ported to an imperative setting, the traditional compilation scheme used for CHR(Prolog) suffers from critical performance issues when executing recursive CHR programs. In our initial JCHR and CCHR systems, the generated code for such programs therefore scaled poorly, as executions constantly failed due to call stack overflows.

The work presented in this chapter constitutes a first analysis of the elementary issues recursion and tail calls in the context of CHR. We demonstrated that the

poor performance of CHR with imperative hosts stems from both:

1. a lack of proper tail call optimisations by typical compilers or interpreters
2. the limited stack space allocated by imperative runtime environments

The former issue even implies many (tail recursive) programs executed with a suboptimal space complexity (linear instead of constant).

We therefore designed and implemented improved compilation schemes for both JCHR and CCHR. Our empirical results confirm our new approach is superior, in particular for compiling to Java. All stack overflow issues are efficiently resolved as follows:

- When applicable, tail call optimisations are performed using trampoline-style execution. This way, optimal stack space complexity is guaranteed, independent of the compiler or interpreter used for the generated code.
- There were needed, our CHR runtime explicitly manages the call stack. Using several additional optimisations, we managed to reduce the resulting overhead to an acceptable level.

In this chapter, we gave a high-level overview of the fundamental techniques used. We moreover outlined how in JCHR2 we extended our earlier ω_r -oriented approach to obtain an effective, more general compilation scheme for (imperative) CHR2 systems. More information on our ω_r -based compilation schemes can be found in (Van Weert, Wuille, Schrijvers, and Demoen 2008); for JCHR an even more detailed, technical description is also given in (Van Weert 2008a).

9.4.1 Related work

Countless (actually most) declarative programming languages have been compiled to imperative host languages, many using similar techniques to manage recursion. We only mention a few that seem most relevant.

Schinz and Odersky (2001) discuss tail call elimination in their Funnel-to-Java compiler. They introduce a trampoline variant that performs ‘higher jumps’: that is, execution only returns to the trampoline after a recursion depth counter has reached a certain threshold; until then regular Java method calls are simply used for tail calls. They list several declarative languages that are compiled to Java, each using the trampoline technique. Our solution is more general as it deals with all instances of (recursive) calls, not tail calls alone.

Of the most prominent, state-of-the-art CHR systems, JCHR and CCHR are currently the only systems that explicitly optimise for recursion. CHR^{TP} sidesteps the issue by only allowing a single active constraint per priority. And, as discussed earlier, in ω_r -based CHR(Prolog) systems the issue is less of a problem because the host’s interpreter mostly adequately copes with recursion. We have

not studied the approach or performance of e.g. CHR(FP) systems. Compilers for functional host languages though are normally similarly designed for an optimal execution of recursive functions.

Most production rule systems (cf. Chapter 2) are Rete-based, and operate using explicit match-recognise-act iterations, where all recursion is inherently broken by returning to a single control loop. Apparent exception is the LEAPS algorithm. Much like our approach, LEAPS essentially manages an explicit stack of continuations, each representing an active fact and a set of iterators. Miranker et al. (1990) and Brant (1993) give very low level descriptions of this algorithm in terms of fact pointers, which was later reconstructed by Batory et al. (1994) and described using far more intelligible, high-level concepts.

9.4.2 Future work

It is well-known that many recursive procedures can be transformed into equivalent iterative forms. The latter can typically be executed far more efficiently.

Example 9.2. The PRIMES program, for instance, contains the following common CHR rule pattern to implement a loop:

```
candidate(1) <=> true.  
candidate(N) <=> N > 1 | prime(N), candidate(N-1).
```

Obviously, this could be transformed into an equivalent iteration of the form:

```
procedure candidate(N)  
  while N > 1  
    ...  
  end  
end
```

While a trampoline-based approach guarantees the correct complexity, the constant factors of the above loop would be significantly better. To transform recursive programs to iterative ones, techniques such as unfolding/folding are typically used, e.g. in functional and logical languages (Burstall and Darlington 1977; Debray 1988). For CHR, such program transformation techniques have only been scarcely researched (Tacchella et al. 2007), and are not yet applied by any implementation.

Chapter 10

Optimising Propagation Rules

*If Beethoven had been killed in a plane crash at the age of 22,
it would have changed the history of music. . . and of aviation.*

— **Tom Stoppard** (born 1937)
British playwright

An important, distinguishing feature of CHR are *propagation rules*. Unlike most rewrite rules, propagation rules do not remove the constraints matched by their head. They only add extra, logically implied constraints. To avoid trivial non-termination, each CHR rule is applied at most once with the same combination of constraints. This requirement stems from the formal study of properties such as termination and confluence (Section 4.3), and is reflected in the operational semantics implemented by almost all systems (Section 4.2).

To prevent reapplication, a standard CHR runtime system maintains a so-called *propagation history*, containing a tuple for each constraint combination that fired a rule. Efficiently implementing a propagation history is challenging. Even with the implementation techniques discussed in Section 10.1, maintaining a propagation history remains expensive. For a given program, the worst-case space complexity of a history is $\mathcal{O}(|\mathcal{S}|^n)$, with $|\mathcal{S}|$ the size of the constraint store, and n the maximum number of occurrences in a (propagation) rule's head (cf. Section 10.1). Clearly, this often dominates the worst-case space complexity of the entire CHR program. Our empirical observations confirm the history often has a significant impact on both space and time performance.

Existing literature on CHR compilation nevertheless pays only scant attention to history-related optimisations. We mend this discrepancy by introducing several novel approaches to history-related performance issues. We show that for almost all CHR rules the propagation history can be eliminated completely, using either

innovative, alternate techniques to prevent rule reapplication (Section 10.2), or by proving that reapplication has no observable effect (Section 10.3). We implemented these optimisations in two state-of-the-art CHR implementations, K.U.Leuven CHR for SWI-Prolog and K.U.Leuven JCHR. Experimental results confirm the relevance and effectiveness of our optimisations (Section 10.4). For those cases where histories are still needed, an efficient history implementation nevertheless remains important. We briefly review existing approaches and some minor contributions of our own in Section 10.1.

All optimisations in this chapter are worked out for CHR systems that implement the refined operational semantics, including formal correctness proofs in the ω_r framework. Section 10.5.2 discusses extensions towards more recent variants and extensions of CHR.

10.1 Propagation History Implementation

It is as stated by Duck (2005, Section 4.3.4):

“The propagation history is very easy to implement naively, but quite challenging to implement efficiently. ”

In this section, we briefly explore the design space for the implementation of propagation histories. We point out the challenges involved, review existing approaches, and conclude with introducing some minor improvements.

Obviously, tuples have to be stored in some efficient data structure, e.g. a balanced tree or a hash table. Naively implemented, tuples are only added to the propagation history, but never removed (as is the case in the formal ω_r semantics; cf. Section 4.2.3). This potentially leads to unbounded memory use.

In terms of our compilation scheme of Section 8.2.2, the main challenge is thus efficiently implementing the `cleanHistory()` operation. All tuples referring to removed constraints are trivially redundant. Formally, for a constraint store \mathbb{S} and a history \mathbb{T} , these are all tuples not in $live(\mathbb{T}, \mathbb{S}) = \{(\rho, I) \in \mathbb{T} \mid I \subseteq ID(\mathbb{S})\}$. Yet even with perfect garbage collection, the history’s worst-case space remains $\mathcal{O}(|\mathbb{S}|^n)$, with n the maximum number of occurrences in a propagation rule’s head. This typically dominates the space complexity of the entire CHR program.

Practice, moreover, shows that eagerly removing redundant tuples after each constraint removal is not feasible due to unreasonable time or space overheads (cf. also Duck 2005). CHR implementations therefore commonly use ad-hoc garbage collection techniques, which may result in excessive memory use in the worst case, but perform adequately in practice.¹

¹In Section 6.3.1.9, we established that for CHR rules with aggregates that require fire-many semantics, `cleanHistory()` must remove more history tuples than just $\mathbb{T} \setminus live(\mathbb{T}, \mathbb{S})$. For such rules, an eager, relatively expensive removal policy seems unavoidable in general. We further discuss history management of such rules in Section 10.5.2.

One technique is to lazily remove redundant tuples during history checks, as used in CHR(HAL) by Duck (2005). A second technique is denoted *distributed propagation history* maintenance (Schrijvers 2005). With this technique, there is no global history data structure. Instead, the runtime representation of each individual CHR constraint contains (a subset of) the history tuples they occur in. When a constraint is removed, the corresponding part of the propagation history is removed as well. Both techniques could easily be combined.

We refer to (Duck 2005; Schrijvers 2005) for more information on the implementation of propagation histories in current CHR systems. Most detailed design choices, however, are not fully covered by these theses. The following example summarises the choices we made in our K.U.Leuven JCHR system.

Example 10.1. By default, JCHR maintains a distributed propagation history using hash tables. Separate tables are kept per rule, and stored only in the active constraint. Checking the history thus entails checking a hash table in all partners constraints whose occurrence is not passive (of course, the tuple object and hash code are only created once). History tuples contain integer constraint identifiers rather than constraint objects to restrict memory leakage. For single-headed rules, a boolean field is used instead of a hash table.

10.1.1 Optimising history maintenance

We now introduce three minor optimisations for distributed propagation history maintenance. More specifically, we start from JCHR's default maintenance scheme outlined in Example 10.1. Our optimisations similarly apply to e.g. global histories as well though. To the best of our knowledge, this is the first time these optimisations for CHR history maintenance are introduced.

Firstly, an `inHistory` check is typically immediately followed by `addToHistory` (cf. Listing 8.1 in Section 8.2.2). These instructions can be merged into one.

Secondly, the arity of history tuples can be reduced as follows:

1. If all but one occurrence is passive (cf. Section 8.3.4.4), the identifier for the active constraint is removed from the tuples.
2. The identifiers for occurrences that are uniquely functionally determined (i.e. with a set semantics functional dependency; cf. Section 5.1.7), and whose matching constraints cannot be removed, are similarly left out.

By itself, the performance gains of such tuple reductions are only marginal. When reduced to zero, however, a simple boolean field in the active constraint suffices. And when reduced to one, tuples no longer have to be created, and considerably more efficient hash table data structures can be used.

Thirdly, for two-headed propagation rules, a similar optimisation is always possible. By default, history tuples for a two-headed rule contain two constraint identifiers. It is however more efficient to simply store, in each constraint, the

identifiers of all partner constraints it fired with whilst active. This avoids the creation of tuples, and allows for more efficient hash tables.

For the last optimisation, care must be taken though when both heads are occurrences of the same constraint, as for instance in the *transitivity* rule of the LEQ program. One possibility is to maintain a separate history per occurrence. A more elegant trick though is to use the negated partner constraint identifier if the the active constraint matches one of the occurrences.

10.2 Non-reactive Propagation Rules

Section 10.2.1 contains a first proper study of what we coined *non-reactive CHR rules*. This is the class of CHR rules that are never matched by a reactivated constraint. As illustrated, a substantial portion of CHR rules is non-reactive. In Section 10.2.2, we prove that the history of certain non-reactive propagation rules can be eliminated, as CHR’s operational semantics ensures these rules are never matched by the same constraint combination. For the remaining non-reactive rules, we introduce an innovative, more efficient technique to prevent rule reapplication in Section 10.2.3, and prove its soundness.

10.2.1 Introduction: from fixed to non-reactive CHR

Non-reactive CHR constraints are never reactivated when built-in constraints are added. Formally:

Definition 10.1. A CHR constraint c/n is *non-reactive* in a program \mathcal{P} under a refined operational semantics ω_r^* (ω_r or any of its refinements: see further) iff for any **Solve** transitions of the form $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle S ++ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ in any ω_r^* -derivation D the set of reactivated constraints $S \subseteq \mathbb{S}$ does not contain constraints of type c/n . A rule $\rho \in \mathcal{P}$ is *non-reactive* iff all constraints that occur in its head are non-reactive in \mathcal{P} .

The most obvious instances are so-called *fixed* constraints.² A CHR constraint predicate c/n is fixed, if, for all runtime constraints c of this predicate, all argument values are always fixed—formally: $\text{vars}(c) \subseteq \text{fixed}(\emptyset)$ (Def. 4.10 in Section 4.2.3). If all constraint arguments are fixed, no rules become applicable when adding built-in constraints. Which CHR constraints are fixed is derived from their mode declarations, or using *groundness analysis* (Schrijvers et al. 2005).

Example 10.2. The FIBBO handler depicted in Listing 10.1, performs a bottom-up computation of all Fibonacci numbers up to a given number. The constraint

²For CHR(Prolog), fixed constraints are often also called *ground* constraints.

Listing 10.1 A handler from (Frühwirth 2005c), referred to here as FIBBO, that performs a bottom-up computation of all Fibonacci numbers up to a given number. All constraint arguments are fixed integer values.

```
:- chr_constraint up_to(+int), fib(+int,+int).
up_to(U) => fib(0,1), fib(1,1).
up_to(U), fib(N-1,M1), fib(N,M2) => N < U | fib(N+1,M1+M2).
```

Listing 10.2 A classic CHR handler (cf. e.g. Frühwirth 2005c) that computes Fibonacci numbers using a top-down computation strategy with memoisation.

```
:- chr_constraint fib(+int,?int).
fib(N,M:1) \ fib(N,M2) <=> M1 = M2.
fib(N,M) => N <= 1 | M = 1.
fib(N,M) => N > 1 | fib(N-1,M1), fib(N-2,M2), M = M1 + M2.
```

declarations³ specify all arguments are fixed instances of the host language’s **int** type (the ‘+’ mode declaration indicates a constraint’s argument is fixed).

In theory, for ω_r , a CHR constraint predicate is non-reactive if and only if it is fixed. This follows from the loose upper bound for the set of reactivated constraints in its **Solve** transition (see Figure 4.2 on page 42). The following example shows why the class of non-reactive constraints should be larger:

Example 10.3. Listing 10.2 lists an alternative Fibonacci handler, this time using a top-down computation strategy with memoisation. The **fib/2** constraint is not fixed, and is typically called with a free (logical) variable as second argument—hence the ‘?’ mode declaration. Reactivating **fib/2** constraints is nevertheless pointless, as there are no guards on its second argument. Adding built-in constraints therefore never results in additional applicable rules.

Using unbound, unguarded arguments to retrieve the outcome of a computation is very common in CHR. The standard ω_r semantics nevertheless allows such constraints to be reactivated. In general, CHR constraints should only be reactivated if additional built-in constraints may cause more guards to become entailed. This is insufficiently specified in ω_r . To address this issue, we reintroduce the concept of *anti-monotonicity*, first defined by Schrijvers and Demoen (2004a):

Definition 10.2. A conjunction of built-in constraints B is *anti-monotone* in a set of variables V iff $\forall B_1, B_2 ((\pi_{vars(B)} \setminus V (B_1 \wedge B_2) \leftrightarrow \pi_{vars(B)} \setminus V (B_1))$
 $\rightarrow ((\mathcal{D}_{\mathcal{H}} \not\models B_1 \rightarrow B) \rightarrow (\mathcal{D}_{\mathcal{H}} \not\models B_1 \wedge B_2 \rightarrow B)))$

³The syntax is based on that of the K.U.Leuven CHR system.

Definition 10.3. A CHR program \mathcal{P} is *anti-monotone* in the i 'th argument of a CHR constraint predicate c/n , iff for every occurrence $c(x_1, \dots, x_i, \dots, x_n)$ in \mathcal{P}^* , the guard of the corresponding rule is anti-monotone in $\{x_i\}$.

In this definition, \mathcal{P}^* denotes the linearised normal form of a program \mathcal{P} (pattern linearisation was introduced in Section 4.1.3).

A CHR program is always anti-monotone in fixed or unguarded constraint arguments. Moreover, several typical built-ins are anti-monotone in their arguments. In Prolog, for instance, $\text{var}(X)$ is anti-monotone in $\{X\}$. Using anti-monotonicity, we now define ω'_r , a slight refinement of ω_r :

Definition 10.4. Let $\text{delay_vars}_{\mathcal{P}}(c)$ denote the set of variables occurring in an (identified) CHR constraint c in which \mathcal{P} is not anti-monotone. Then ω'_r is obtained from ω_r by replacing the upper bound on the set of reactivated constraints S in its *Solve* transition with “ $\forall c \in S : \text{delay_vars}_{\mathcal{P}}(c) \not\subset \text{fixed}(\mathbb{B})$ ”.

In Appendix B, we provide a formal proof that ω'_r is indeed an instance of ω_r . The appendix also clarifies the difference with the similar yet incorrect formulation by Schrijvers and Demoen (2004a).

Most rules in general-purpose CHR programs are non-reactive under ω'_r . Several CHR systems, including the K.U.Leuven CHR and JCHR systems, already implement this refinement of ω_r (see Section 8.3.5.2). In the next two subsections, we prove that for non-reactive CHR rules the expensive maintenance of a propagation history can always be avoided.

10.2.2 Propagation history elimination

Because non-reactive CHR constraints are only active once, non-reactive propagation rules often do not require a history:

Example 10.4. The `sum/2` constraint in Listing 10.3 computes the sum of a client's account balances using a common CHR programming idiom to compute aggregates. Similar code is also generated by the source-to-source transformations used to implement aggregates in (Van Weert et al. 2008). The idiom uses a (typically non-reactive) propagation rule to generate a number of constraints, from which, after simplification to a single constraint, the result can be retrieved.

In the example, when the active `gen/1` constraint considers the *generate* rule, it iterates over candidate `account/2` partner constraints. Assuming this iteration is *duplicate-free* (we discuss this property shortly), the *generate* rule never fires with the same constraint combination under ω_r , even if no propagation history is maintained. Indeed, the *generate* rule only adds `sum/1` constraints, which, as there is no `get/1` constraint yet in the store (the body of the `sum_balances` rule is executed from left to right), only fire the *simplify* rule.

Listing 10.3 CHR rules computing the sum of the account balances of a client. These rules may be part of some larger CHR-based banking application.

```

:- chr_constraint account(+client, +float), sum(+client, ?float).
:- chr_constraint gen(+client), sum(+float), get(?float).

sum_balances @ sum(C, Sum) ⇔ gen(C), get(Sum).
generate @ gen(C), account(C,B) ⇒ sum(B).
simplify @ sum(B1), sum(B2) ⇔ sum(B1 + B2).
retrieve @ get(Q), gen(_), sum(Sum) ⇔ Q = Sum.

```

A common mistake is to assume that the history is thus superfluous for all non-reactive CHR rules. The following example shows this is not the case:

Example 10.5. Reconsider the FIBBO handler of Listing 10.1. If an `up_to(U)` constraint is activated, the first rule adds two `fib/2` constraints. Next, the second rule propagates all required `fib/2` constraints, each time with an active `fib/2` constraint. After this, control returns to the suspended `up_to(U)` constraint, and advances to its second occurrence. Some mechanism is then required to prevent the second (non-reactive) propagation rule to add erroneous `fib/2` constraints.

So, non-reactive propagation rules can match the same constraint combination more than once. This occurs if one or more partner constraints for an active constraint in rule ρ were added by firing ρ or some earlier rule with the same active constraint. We say these partner constraints *observe* the corresponding occurrence of the active constraint in ρ . This is similar to the notion of observation discussed earlier in Section 8.3.3.1 in the context of late store optimisations. Formally:

Definition 10.5. Let the k 'th occurrence of a rule ρ 's head be the j 'th occurrence of constraint predicate c/n . Then this occurrence is *unobserved* under a refined operational semantics ω_r^* iff for all **Activate** or **Default** transitions of the form⁴:

$$\langle \mathbb{A}_0, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_- \mapsto_{\rho} \langle [c\#i:j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_-$$

($\mathbb{A}_0[1] = c\#i$ or $\mathbb{A}_0[1] = c\#i:j-1$) the following holds: $\forall(\rho, I) \in \mathbb{T} : I[k] \neq i$, and similarly for all transition sequences starting with a **Propagate** transition

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_- \mapsto_{\rho} \langle B \text{ ++ } \mathbb{A}, \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_- \mapsto_{\rho}^* \langle \mathbb{A}, \mathbb{S}'', \mathbb{B}'', \mathbb{T}'' \rangle_-$$

with $\mathbb{A}[1] = c\#i:j$, $\forall(\rho, I) \in \mathbb{T}'' \setminus \mathbb{T}' : I[k] \neq i$.

Let ω_r^\dagger denote the semantics obtained from ω_r' by adding the following condition to its **Propagate** and **Simplification** transitions: “If the j 'th occurrence of c is unobserved under ω_r' , then $\mathbb{T}' = \mathbb{T}$ ”. Also, to prevent trivial reapplication in a consecutive sequence of **Propagate** transitions (see e.g. Example 10.4), propagation in ω_r^\dagger is defined to be *duplicate-free*:

⁴We use ‘ $_$ ’ to denote that we are not interested in the identifier counter.

Definition 10.6 (Duplicate-free Propagation). Propagation in a refined operational semantics ω_r^* is *duplicate-free* iff for all ω_r^* -derivations D of a CHR program \mathcal{P} where the j 'th occurrence of c is kept, the following holds:

$$\text{if } \left\{ \begin{array}{l} \sigma_1 \mapsto_{\mathcal{P}} \sigma_2 \mapsto_{\mathcal{P}}^* \sigma'_1 \mapsto_{\mathcal{P}} \sigma'_2 \text{ is part of } D \\ \sigma_1 = \langle [c\#i:j|\mathbb{A}], \mathbb{S}, \dots \rangle_{-} \text{ and } \sigma'_1 = \langle [c\#i:j|\mathbb{A}], \mathbb{S}', \dots \rangle_{-} \\ \sigma_1 \mapsto_{\mathcal{P}} \sigma_2 \text{ is a } \mathbf{Propagate} \text{ transition applied with constraints } H \subseteq \mathbb{S} \\ \sigma'_1 \mapsto_{\mathcal{P}} \sigma'_2 \text{ is a } \mathbf{Propagate} \text{ transition applied with constraints } H' \subseteq \mathbb{S}' \\ \text{between } \sigma_2 \text{ and } \sigma'_1 \text{ no } \mathbf{Default} \text{ transition occurs of the form} \\ \sigma_2 \mapsto_{\mathcal{P}}^* \langle [c\#i:j|\mathbb{A}], \dots \rangle_{-} \mapsto_{\mathcal{P}} \langle [c\#i:j+1|\mathbb{A}], \dots \rangle_{-} \mapsto_{\mathcal{P}}^* \sigma'_1 \end{array} \right.$$

then $H \neq H'$.

Before we link this with the notion of duplicate-free constraint iterators from Section 8.2.1.2, we first proof the soundness of eliminating the history of unobserved CHR rules, by showing ω_r^\dagger and ω_r^* are equivalent:

Theorem 10.1. *Define the mapping function α^\dagger as follows:*

$$\alpha^\dagger(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \{(\rho, I) \in \mathbb{T} \mid \rho \text{ is not unobserved}\} \rangle_n$$

If D is an ω_r^ derivation, then $\alpha^\dagger(D)$ is an ω_r^\dagger derivation. Conversely, if D is an ω_r^\dagger derivation, then there exists an ω_r^* derivation D' such that $\alpha^\dagger(D) = D'$.*

Proof. If D is an ω_r^* derivation, then $\alpha^\dagger(D)$ is clearly an ω_r^\dagger derivation.

For the reverse direction, let D be an ω_r^\dagger derivation, and D' the derivation obtained from D by adding the necessary tuples to the propagation history. That is, for each **Propagate** or **Simplify** transition in D of the form

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \text{ ++ } \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \rangle_n$$

the corresponding transition in D' becomes of the form

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \text{ ++ } \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \cup \{(\rho, I)\} \rangle_n \quad (10.1)$$

All **Propagate** and **Simplify** transitions in D' now have form (10.1). It suffices to show that for all these transitions $(\rho, I) \notin \mathbb{T}$ (note that we used \cup and not \sqcup , as the disjointness of the union is exactly what still needs to be shown).

Suppose there exist transitions of form (10.1) in D' for which $(\rho, I) \in \mathbb{T}$. Without loss of generality, we may consider the first such transition. Suppose the active constraint $c\#i:j$ matched the k 'th occurrence in ρ 's head. Then, clearly, this occurrence must be unobserved, and D' starts with the transition sequence

$$\begin{aligned} D[1] &\mapsto_{\mathcal{P}}^* \langle \mathbb{A}_0, \mathbb{S}_0, \mathbb{B}_0, \mathbb{T}_0 \rangle_{n_0} && (\mathbb{A}_0[1] \neq c\#i:j) \\ &\mapsto_{\mathcal{P}} \langle [c\#i:j|\mathbb{A}'], \mathbb{S}_0, \mathbb{B}_0, \mathbb{T}_0 \rangle_{n_0} \\ &\mapsto_{\mathcal{P}}^* \langle [c\#i:j|\mathbb{A}'], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \\ &\mapsto_{\mathcal{P}} \langle B \text{ ++ } [c\#i:j|\mathbb{A}'], \mathbb{S}', \mathbb{B}, \mathbb{T} \cup \{(\rho, I)\} \rangle_n \end{aligned}$$

with $[c\#i:j|\mathbb{A}'] = \mathbb{A}$, $I[k] = i$ and $(\rho, I) \in \mathbb{T}$. The sequence of transitions without this last invalid **Propagate** transition is the beginning of a valid ω'_r derivation. Therefore, by Definition 10.5, $(\rho, I) \notin \mathbb{T}_0$ (the j 'th occurrence of c is unobserved, and $I[k] = i$). Prior to the invalid **Propagate** transition, the non-reactive $c\#i:j$ active constraint repeatedly appears on top of the activation stack in a sequence of zero or more **Propagate** transitions:

$$\begin{aligned} \dots \mapsto_{\mathcal{P}} \langle [c\#i:j|\mathbb{A}], \mathbb{S}_0, \mathbb{B}_0, \mathbb{T}_0 \rangle_{n_0} &\mapsto_{\mathcal{P}} \langle B_1 ++ [c\#i:j|\mathbb{A}], \mathbb{S}'_0, \mathbb{B}_0, \mathbb{T}_0 \cup \{(\rho, I_1)\} \rangle_{n_0} \\ &\mapsto_{\mathcal{P}}^* \langle [c\#i:j|\mathbb{A}], \mathbb{S}_1, \mathbb{B}_1, \mathbb{T}_1 \rangle_{n_1} \\ &\mapsto_{\mathcal{P}} \langle B_2 ++ [c\#i:j|\mathbb{A}], \mathbb{S}'_1, \mathbb{B}_1, \mathbb{T}_1 \cup \{(\rho, I_2)\} \rangle_{n_1} \\ &\dots \\ &\mapsto_{\mathcal{P}} \langle [c\#i:j|\mathbb{A}], \mathbb{S}_m, \mathbb{B}_m, \mathbb{T}_m \rangle_{n_m} \end{aligned}$$

with $\mathbb{S}_m = \mathbb{S}$, $\mathbb{B}_m = \mathbb{B}$, $\mathbb{T}_m = \mathbb{T}$, and $n_m = n$. By Definition 10.5:

$$\forall i' \in [1, m] : \forall (\rho, I') \in \mathbb{T}_{i'} \setminus (\mathbb{T}_{i'-1} \cup \{(\rho, I_{i'})\}) : I'[k] \neq i'$$

and consequently (by induction):

$$\Theta = \{(\rho, I') \in \mathbb{T} \setminus \mathbb{T}_0 \mid I'[k] = i\} = \{(\rho, I_1), \dots, (\rho, I_m)\}$$

Because propagation is duplicate-free, $(\rho, I) \notin \Theta$. Consequently, as $I[k] = i$, $(\rho, I) \notin \mathbb{T} \setminus \mathbb{T}_0$. And we already showed that $(\rho, I) \notin \mathbb{T}_0$, so $(\rho, I) \notin \mathbb{T}$. By contradiction, no (first) transition of form (10.1) can exist with $(\rho, I) \in \mathbb{T}$. \square

Implementation

In the terminology of Section 8.2.1.2, most constraint iterators used in CHR implementations are strongly duplicate-free. For such iterators, the requirement of Definition 10.6 is directly met. For those iterators that are only weakly duplicate-free, a temporary table of already seen identifiers can be kept as long as the iterator is in use.

The main difficulty in the implementation of this optimisation though is deriving that a rule is unobserved. The abstract interpretation-based late storage analysis of Schrijvers, Stuckey, and Duck (2005) can be adapted for this purpose. We discussed this analysis that derives a similar observation property in Section 8.3.3.1. The details are beyond the scope of this section.

10.2.3 Optimised reapplication avoidance

Non-reactive CHR rules that are not unobserved, such as the second rule in the FIBBO handler of Example 10.5, do require some mechanism to prevent reapplication. Moreover, even if a rule is unobserved, this does not mean the compiler is capable of deriving it. In this section we therefore present a novel, very

efficient technique that prevents the reapplication of any non-reactive propagation rule without maintaining a costly history.

The central observation is that, when a non-reactive rule is applied, the active constraint is always more recent than its partner constraints:

Lemma 10.2. *Let \mathcal{P} be an arbitrary CHR program, with $\rho \in \mathcal{P}$ a non-reactive rule, and D an arbitrary ω'_r derivation with this program. Then for each **Simplify** or **Propagate** transition in D of the form*

$$\langle [c\#i:j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T} \sqcup \{(\rho, I_1 ++ [i] ++ I_2)\} \rangle_n \quad (10.2)$$

the following holds: $\forall i' \in I_1 \cup I_2 : i' < i$.

Proof. Assume $i' = \max(I_1 \sqcup I_2)$ with $i' \geq i$. By CHR's definition of matching substitutions (Definition 4.3), $i' \neq i$, and $\exists c'\#i' \in \mathbb{S}$. This $c'\#i'$ partner constraint must have been stored in an **Activate** transition. Since $i' = \max(I_1 \sqcup \{i\} \sqcup I_2)$, in D , this transition came *after* the **Activate** transitions of all other partners, including $c\#i$. In other words, all constraints in the matching combination of transition (10.2) were stored prior to the activation of $c'\#i'$. Also, in (10.2), $c\#i$ is back on top of the activation stack. Because c is non-reactive, and thus never put back on top by a **Reactivate** transition, the later activated $c'\#i'$ must have been removed from the stack in a **Drop** transition. This implies that all applicable rules matching c' must have fired. As all required constraints were stored (see earlier), this includes the application of ρ in (10.2). By contradiction, our assumption is false, and $i' < i$. \square

Let ω_r^\ddagger denote the semantics obtained from ω'_r by replacing the propagation history condition in its **Simplify** and **Propagate** transitions with:

If ρ is non-reactive, then $\forall i' \in \text{ID}(H_1 \cup H_2) : i' < i$ and $\mathbb{T}' = \mathbb{T}$. Otherwise, let $t = (\rho, \text{ID}(H_1) ++ [i] ++ \text{ID}(H_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

Propagation in ω_r^\ddagger is again duplicate-free, as defined by Definition 10.6. Similarly to Theorem 10.1, the following theorem proves that ω'_r and ω_r^\ddagger are equivalent:

Theorem 10.3. *Define the mapping function α^\ddagger as follows:*

$$\alpha^\ddagger(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \{(\rho, I) \in \mathbb{T} \mid \rho \text{ is a reactive CHR rule}\} \rangle_n$$

If D is an ω'_r derivation, then $\alpha^\ddagger(D)$ is an ω_r^\ddagger derivation. Conversely, if D is an ω_r^\ddagger derivation, then there exists an ω'_r derivation D' such that $\alpha^\ddagger(D) = D'$.

Proof. If D is an ω'_r derivation, then, by Lemma 10.2, $\alpha^\ddagger(D)$ is an ω_r^\ddagger derivation.

For the reverse direction, let D be an ω_r^\ddagger or ω_r^\ddagger derivation, and D' obtained from D by adding the necessary tuples to the propagation history, as in the proof of Theorem 10.1. I.e., all **Propagate** and **Simplify** transitions in D' have form (10.1). It again suffices to show that for all these transitions $(\rho, I) \notin \mathbb{T}$.

```

procedure up_to(U)#ID : 2
foreach fib(N,M2)#ID2 in ...
  foreach fib(N-1,M1)#ID1 in ...
    if N < U
      if ID < ID1 and ID < ID2
        ⋮

```

(a) Efficient reapplication avoidance using identifier comparisons

```

procedure up_to(U)#ID : 2
foreach fib(N,M2)#ID2 in ...
  if ID < ID2 and N < U
    foreach fib(N-1,M1)#ID1 in ...
      if ID < ID1
        ⋮

```

(b) After loop-invariant code motion

Figure 10.1: Optimised reapplication avoidance for non-reactive rules. These listings show pseudocode for the second occurrence of the `up_to/1` constraint in the FIBBO program of Listing 10.1.

First, we show that Lemma 10.2 still holds for the derivation D . That is, for all transitions of D of form (10.1), if the active constraint matched the k 'th occurrence in ρ 's head, then $I[k] = \max(I)$. By definition of ω_r^\dagger , this is true for the tuples that were not added to the history in the original derivation D . For those added in both D and D' , this also holds by definition of ω_r^\dagger and Lemma 10.2.

Suppose, for some transition of form (10.1), that $(\rho, I) \in \mathbb{T}$, and that the active constraint matched the k 'th occurrence of ρ . Then $I[k] = \max(I)$. Moreover, when the (ρ, I) tuple was first added to the history, by uniqueness of constraint identifiers, the active constraint was the same constraint as active in the considered constraint. As propagation is duplicate-free in D , and the active constraint is non-reactive, this is not possible. \square

Implementation

As duplicate-freeness is easily enforced as before in Section 10.2.2, implementing identifier-based reapplication prevention is trivial.

Example 10.6. Figure 10.1(a) shows the generated code for the second occurrence of the `up_to/1` constraint in Listing 10.1. For the query `up_to(U)`, the propagation history for the corresponding rule would require $\mathcal{O}(U)$ space. Because all constraints are non-reactive, however, no propagation history has to be maintained. Simply comparing constraint identifiers suffices.

In Section 8.3.2.1 we discussed *loop-invariant code motion* optimisation, an optimisation aimed at checking guards and other tests as soon as possible in the nested loops. Contrary to a propagation history check, identifier comparisons are eligible for code motion, as illustrated in Figure 10.1(b). This can prune the search space of candidate partner constraints considerably.

In fact, Lemma 10.2 does not only apply to propagation rules, but also to simplification and simpagation rules. Whilst maintaining a history for non-

propagation rules is pointless, comparing partner constraint identifiers in outer loops is not, as they can avoid redundant iterations of nested loops.

10.3 Idempotence

Constraints in CHR handlers that specify traditional constraint solvers typically range over unbound variables, and are thus highly reactive. Without a history, constraint reactivations are likely to cause reactive propagation rules to fire multiple times with the same combination. For constraint solvers, however, such additional rule applications generally have no effect, as most ‘pure’ constraints have set semantics. This was discussed in detail in Section 5.1.6. A stronger property, called *idempotence*, than set semantics is required though. For idempotent rules, we can prove that the propagation history may be eliminated as well. Because the history is a relatively expensive data structure, this can considerably improve performance.

Example 10.7. Suppose for the classic LEQ program of Listing 4.1 that the (reactive) *transitivity* propagation rule is allowed to fire a second time with the same constraint combination matching its head, thus adding a $\text{leq}(X, Z)$ constraint for the second time. If the earlier told duplicate is still in the store, this redundant $\text{leq}(X, Z)$ constraint is immediately removed by the *idempotence* rule. Otherwise, the former duplicate must have been removed by either the *reflexivity* or the *antisymmetry* rule. It is easy to see that in this case $X = Z$, which implies the new, redundant $\text{leq}(X, Z)$ constraint is again removed immediately by the *reflexivity* rule.

We say the $\text{leq}/2$ constraint of the above example is *idempotent*. With $\text{live}(\mathbb{T}, \mathbb{S})$ defined as in Section 10.1, idempotence is formally defined as:

Definition 10.7. A CHR constraint predicate c/n is *idempotent* in a CHR program \mathcal{P} under a refined semantics ω_r^* iff for any state $\sigma = \langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ in a ω_r^* derivation D with c a CHR constraint, the following holds: if earlier in D a state $\langle [c'|\mathbb{A}'], \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_{n'}$ occurs with $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow c = c'$, then $\sigma \xrightarrow{\mathcal{P}}^* \langle \mathbb{A}, \mathbb{S}'', \mathbb{B}'', \mathbb{T}'' \rangle_{n''}$ with $\mathbb{S}'' = \mathbb{S}$, $\text{live}(\mathbb{T}'', \mathbb{S}) = \text{live}(\mathbb{T}, \mathbb{S})$, and $\mathcal{D}_{\mathcal{H}} \models \pi_{\text{vars}(\mathbb{B}) \cup \text{vars}(D[1])}(\mathbb{B}'') \leftrightarrow \mathbb{B}$.

In words, an idempotent constraint c for which a syntactically equal constraint c' was told earlier in the same derivation, is removed without making any observable state change. Since ‘ $\xrightarrow{\mathcal{P}}^*$ ’ denotes a finite derivation, telling duplicate idempotent CHR constraints does not affect termination either.

We do not consider arbitrary, extra-logical host language statements here, and assume all built-in constraints b are idempotent, that is: $\forall b : \mathcal{D}_{\mathcal{H}} \models b \wedge b \leftrightarrow b$. By adding “If $\mathcal{D}_{\mathcal{H}} \models (\mathbb{B} \wedge b) \leftrightarrow \mathbb{B}$, then $S = \emptyset$ ” to the **Solve** transition of ω_r (or any of its refinements in Section 10.2), we avoid redundant constraint reactivations

when idempotent built-in constraints are told. This is correct, as **Solve**'s upper bound on S already specifies that any matching that was already possible prior to b 's addition may be omitted from S . Most CHR systems implement this optimisation. Denote the resulting semantics ω_r^{idem} .

Definition 10.8. A CHR rule $\rho \in \mathcal{P}$ is *idempotent* (under ω_r^{idem}) iff all CHR constraint predicates that occur in its body are idempotent in \mathcal{P} .

We now prove that an idempotent propagation rule may be fired more than once with the same combination of constraints, without affecting a program's operational semantics. Let $\omega_r^{idem'}$ denote the semantics obtained by adding the following phrase to the **Simplify** and **Propagate** transitions of ω_r^{idem} :

If the rule ρ is idempotent, then $\mathbb{T}' = \mathbb{T}$; otherwise, ... (as before)

Assuming then that propagation for $\omega_r^{idem'}$ is duplicate-free⁵ in the sense of Definition 10.6, the $\omega_r^{idem'}$ semantics is equivalent to ω_r^{idem} . More precisely:

Theorem 10.4. *If D' is an $\omega_r^{idem'}$ derivation, then there exists an ω_r^{idem} derivation D with $D[1] = D'[1]$ such that a monotonic function α can be defined from the states in D to states in D' for which*

- $\alpha(D[1]) = D'[1]$
- if $\alpha(D[i]) = D'[k]$ and $\alpha(D[j]) = D'[l]$ with $i < j$, then $k < l$
- if $\alpha(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_{n'}$ then $\mathcal{D}_{\mathcal{H}} \models \pi_{\text{vars}(\mathbb{B}) \cup \text{vars}(D[1])}(\mathbb{B}') \leftrightarrow \mathbb{B}$, $\mathbb{A}' = \mathbb{A}$, $\mathbb{S}' = \mathbb{S}$, and $\text{live}(\mathbb{T}', \mathbb{S}) = \text{live}(\mathbb{T}, \mathbb{S}) \setminus \{(\rho, I) \in \mathbb{T} \mid \rho \text{ is idempotent}\}$.

Conversely, if D is an ω_r^{idem} derivation, then an $\omega_r^{idem'}$ derivation D' exists with $D'[1] = D[1]$ for which a function with the same properties can be defined.

Proof sketch. An $\omega_r^{idem'}$ derivation D' only differs from the corresponding ω_r^{idem} derivation D when a **Propagate** transition fires an idempotent propagation rule ρ using a combination of constraints that fired ρ before. This $\omega_r^{idem'}$ transition has form $\sigma_0 = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B ++ \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n = \sigma_1$. Because ρ 's body B is idempotent, it follows from Definition 10.7 that the remainder of D' begins with $\sigma_1 \mapsto_{\mathcal{P}}^* \sigma'_0 = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}', \mathbb{T}' \rangle_{n'}$, with $\mathcal{D}_{\mathcal{H}} \models \pi_{\text{vars}(\mathbb{B}) \cup \text{vars}(D[1])}(\mathbb{B}') \leftrightarrow \mathbb{B}$, and $\text{live}(\mathbb{T}', \mathbb{S}) = \text{live}(\mathbb{T}, \mathbb{S})$. Because σ'_0 is thus essentially equivalent to σ_0 , we simply omit states σ_1 to σ'_0 in the corresponding ω_r^{idem} derivation D .

Given above observations it is straightforward to construct the mapping function α and the required derivations for both directions of the proof. \square

For multi-headed propagation rules, reapplication is often cheaper than maintaining and checking a propagation history. The experimental results of Section 10.4 confirm this. Still, heuristics should probably be used though to estimate the cost of reapplication versus the cost of maintaining a history.

⁵In this case a *finite* number of duplicate propagations would also not be a problem. Consequently, weakly duplicate-free iterators, for instance, are sufficient (cf. Section 8.2.1.2).

10.3.1 Deriving idempotence

The main challenge lies in automatically deriving that a CHR constraint is idempotent. A wide class of idempotent CHR constraints should be covered:

Example 10.8. Many CHR-based constraint solvers contain a rule such as:

$$\text{in}(X, L_1, U_1) \setminus \text{in}(X, L_2, U_2) \Leftrightarrow L_2 \leq L_1, U_2 \geq U_1 \mid \text{true}.$$

Here, ‘ $\text{in}(X, L, U)$ ’ denotes that the variable X lies in the interval $[L, U]$. The $\text{in}/3$ constraint is probably idempotent (it depends on the preceding rules). There is however an important difference with the $\text{leq}/2$ constraint in Example 10.7: by the time the constraint is told for the second time, the earlier told duplicate may now be replaced with a *syntactically different* constraint—in this case: a constraint representing a smaller interval domain.

Theorem 10.5 provides a sufficiently strong syntactic condition for determining the idempotence of a CHR constraint. It uses arbitrary pre-orders on the constraint’s arguments. For the three arguments of the $\text{in}/3$ constraint in Example 10.8 for instance, the pre-orders $=$, \leq and \geq can be used respectively.

Let \mathcal{P}^* again denote the linearised normal form of a CHR program \mathcal{P} , and $bi(B)$ and $chr(B)$ the conjunction of built-in respectively CHR constraints that occur in a constraint conjunction B . Then:

Theorem 10.5. *A CHR constraint predicate c/n is idempotent in \mathcal{P} under ω_r^{idem} if for a series of pre-orders $\triangleleft_1, \dots, \triangleleft_n$:*

1. *There exists a rule of the form “ $c(y_1, \dots, y_n) \setminus c(x_1, \dots, x_n) \Leftrightarrow G \mid \text{true}$.” in \mathcal{P}^* with $\mathcal{D}_{\mathcal{H}} \models (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n) \rightarrow G$.*

Let ρ be the first such rule occurring in the \mathcal{P}^ sequence.*

2. *All rules in \mathcal{P}^* prior to ρ that contain an occurrence of c/n have a trivial body ‘ true ’, and do not contain any removed occurrences apart from possibly that c/n occurrence.*

These first two conditions ensure that newly told duplicate c/n constraints are always removed without adding or removing any constraints, if either the original duplicate is still alive, or if it has been replaced with a ‘smaller’ version. The third condition deals with the final remaining case where the original duplicate is not replaced with a ‘smaller’ version, but truly removed by simplification:

3. *Consider a set of n mutually distinct variables $V = \{x_1, \dots, x_n\}$. Define $\Phi = \pi_V(G \wedge bi(B))$ for all removed occurrences of c/n in a rule of \mathcal{P}^* that can be renamed to the form ‘ $H_k \setminus H_{r_1}, c(x_1, \dots, x_n), H_{r_2} \Leftrightarrow G \mid B$ ’ (H_k, H_{r_1} , and H_{r_2} may be empty), such that $\neg \exists c(y_1, \dots, y_n) \in H_k \cup chr(B) : \mathcal{D}_{\mathcal{H}} \models G \wedge bi(B) \rightarrow (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n)$. For each of these occurrences, either $\mathcal{D}_{\mathcal{H}} \models \Phi \leftrightarrow \text{false}$, or the following two conditions hold:*

- (a) *There exists a rule in \mathcal{P}^* that can be renamed to the form ‘ $c(x_1, \dots, x_n) \Leftrightarrow G \mid B$ ’, such that $bi(B) = B$ and $\mathcal{D}_{\mathcal{H}} \models \Phi \rightarrow (G \wedge B)$. Let ρ' be the first such rule occurring in the \mathcal{P}^* sequence.*
- (b) *All rules in \mathcal{P}^* prior to ρ' that contain an occurrence of c/n can be renamed to “ $H_k \setminus H_r \Leftrightarrow G \mid B$ ” with $H_k \uparrow\uparrow H_r = H_1 \uparrow\uparrow [c(x_1, \dots, x_n)] \uparrow\uparrow H_2$, such that either*
- $\mathcal{D}_{\mathcal{H}} \models \Phi \rightarrow \neg G$; or
 - $H_r \subseteq [c(x_1, \dots, x_n)] \wedge (bi(B) = B) \wedge \mathcal{D}_{\mathcal{H}} \models (\Phi \wedge G) \rightarrow B$; or
 - $\exists c(y_1, \dots, y_n) \in H_1 \cup H_2 : \mathcal{D}_{\mathcal{H}} \models (\Phi \wedge G) \rightarrow (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n)$.

Proof. Let \mathcal{P} be an arbitrary CHR program, and suppose that in this program the four conditions listed in Theorem 10.5 hold for CHR constraint c/n . Let D be an arbitrary ω_r^{idem} derivation of \mathcal{P} of the form

$$\langle Q, \emptyset, true, \emptyset \rangle_1 \xrightarrow{\star}_{\mathcal{P}} \sigma' = \langle [c' \mid \mathbb{A}'], \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle'_n \xrightarrow{\star}_{\mathcal{P}} \sigma = \langle [c \mid \mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$$

with c and c' CHR constraint of type c/n such that $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow c = c'$. Clearly, $D \ni \sigma' \xrightarrow{\star}_{\mathcal{P}} \langle [c' \# n' \mid \mathbb{A}], \{c' \# n'\} \cup \mathbb{S}', \mathbb{B}', \mathbb{T}' \rangle_{n'+1}$, that is: at some point earlier in the derivation the duplicate constraint $c' \# n'$ was added to the constraint store. Suppose now that $\mathcal{D}_{\mathcal{H}} \models \pi_{\emptyset}(\mathbb{B})$ (the case with $\mathcal{D}_{\mathcal{H}} \models \neg \pi_{\emptyset}(\mathbb{B})$ is straightforward), and that $\sigma \xrightarrow{\star}_{\mathcal{P}} \sigma_0 = \langle [c \# n \mid \mathbb{A}], \{c \# n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1} \xrightarrow{\star}_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}'', \mathbb{B}'', \mathbb{T}'' \rangle_{n''}$. Then it remains to be shown that $\mathbb{S}'' = \mathbb{S}$, $\mathcal{D}_{\mathcal{H}} \models \pi_{vars(\mathbb{B}) \cup vars(D[1])}(\mathbb{B}'') \leftrightarrow \mathbb{B}$, and $live(\mathbb{T}, \mathbb{S}) = live(\mathbb{T}'', \mathbb{S}'')$. For the derivation starting from σ_0 , with active constraint $c \# n$, we consider two cases. Let \triangleleft denote the pre-order on c/n constraints defined as $c(x_1, \dots, x_n) \triangleleft c(y_1, \dots, y_n) \leftrightarrow x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n$, then:

Case A $\exists c'' \# n'' \in \mathbb{S} : \mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow c \triangleleft c''$

Let ρ be the rule as defined in condition 1. Then $appl(\rho, [c'' \# n''], [c \# n], \mathbb{B})$, so the active constraint $c \# n$ is removed when it reaches rule ρ . By condition 2, none of the rules that are applied with $c \# n$ active before it is removed (by an application of ρ or earlier) remove or add any constraints. Therefore, no other CHR constraint becomes active before $c \# n$ is removed, and $\sigma_0 \xrightarrow{\star}_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}, \mathbb{B}'', \mathbb{T}'' \rangle_{n''}$ with $\pi_{vars(\mathbb{B}) \cup vars(D[1])}(\mathbb{B}'') \leftrightarrow \mathbb{B}$ (only matching substitutions may be added to \mathbb{B}), and $live(\mathbb{T}, \mathbb{S}) = live(\mathbb{T}'', \mathbb{S})$ (all rules fired involve $c \# n$).

Case B $\neg \exists c'' \# n'' \in \mathbb{S} : \mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow c \triangleleft c''$

Given this assumption, the $c' \# n'$ constraint must have been removed from the store at some point in the derivation. Moreover, this occurred when matching an occurrence that, after renaming, had form “ $H_k \setminus H_{r_1}, c(x_1, \dots, x_n), H_{r_2} \Leftrightarrow G \mid B$ ”. If for this rule the following holds:

	SWI		JCHR	
	tree	2-hash	hash	2-hash
EQ(35)	3,465	N/A ⁶	47	37 (79%)
LEQ(70)	3,806	2,866 (75%)	85	65 (76%)

Table 10.1: Benchmark results (in average milliseconds) for two-headed rules.

$$\exists c(y_1, \dots, y_n) \in H_k \cup \text{chr}(B) : \mathcal{D}_{\mathcal{H}} \models G \wedge \text{bi}(B) \rightarrow (x_1 \triangleleft_1 y_1 \wedge \dots \wedge x_n \triangleleft_n y_n) \quad (10.3)$$

then a constraint $c' \# n''$ with $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow c = c' \triangleleft c''$ existed in, or was added to the store \mathbb{S} at the moment $c' \# n'$ was removed. This constraint, in turn, must have been removed, so we apply the above reasoning again, only this time for $c' \# n''$ instead of $c' \# n'$.

After a finite number of times, $c' \# n'$, or one of its derived constraints $c' \# n''$, must have been removed by a rule for which formula (10.3) does not hold. For $V = \{x_1, \dots, x_n\}$, define $\Phi = \pi_V(G \wedge \text{bi}(B))$ as in the Theorem. The case where $\mathcal{D}_{\mathcal{H}} \models \Phi \leftrightarrow \text{false}$ is trivial, so assume $\mathcal{D}_{\mathcal{H}} \models \Phi \not\leftrightarrow \text{false}$. Then $\mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \Phi$, and conditions 3(a) and (b) hold. By condition 3(a), $\exists \rho' \in \mathcal{P}^* : \text{appl}(\rho', \emptyset, [c \# n], \Phi)$, and thus $\exists \rho' \in \mathcal{P}^* : \text{appl}(\rho', \emptyset, [c \# n], \mathbb{B})$. This rule ρ' furthermore does not add CHR constraints, nor does it add built-in constraints that are not already entailed by \mathbb{B} . Condition 3(b) then ensures that none of the rules applied whilst $c \# n$ is active, and before the c constraint is removed (by an application of ρ' or earlier), remove or add CHR constraints, or add untailed built-in constraints. Therefore, analogously to Case A, $\sigma_0 \mapsto_{\mathcal{P}}^* \langle \mathbb{A}, \mathbb{S}, \mathbb{B}'', \mathbb{T}'' \rangle_{n''}$ with $\text{live}(\mathbb{T}'', \mathbb{S}) = \text{live}(\mathbb{T}, \mathbb{S})$ and $\mathcal{D}_{\mathcal{H}} \models \pi_{\text{vars}(\mathbb{B}) \cup \text{vars}(D[1])}(\mathbb{B}'') \leftrightarrow \mathbb{B}$. \square

10.4 Evaluation

We implemented the optimisations introduced in this paper in the K.U.Leuven CHR system for SWI-Prolog, and in the K.U.Leuven JCHR system for Java, and evaluated them using standard CHR benchmarks and constraint solvers.

Table 10.1 shows some empirical results measured for our improved history implementation for two-headed rules. The new hash tables are about 25% faster. Since the K.U.Leuven CHR system normally maintains the history using a balanced tree, the hash table actually improves the time complexity.

More important though are the benchmark timings for our history elimination optimisations. These are given in Tables 10.2 and 10.3. In both tables, the *history*

⁶In the current SWI implementation, the history of a two-headed propagation rule is only optimised if there are no other propagation rules in the program. In JCHR, this is irrelevant, as JCHR maintains a separate history per rule.

	SWI		JCHR			total # rules	n-head. prop.				
	history	non-react	history	non-react	non-react+		1	2	3	>	
FIBBO(1000)	15,929	4,454 (28%)	70	67 (95%)	21 (30%)	3	1	-	1	-	
FIBBO(3000)	<i>timeout</i>	<i>timeout</i>	542	464 (86%)	153 (28%)	3	1	-	1	-	
FLOYD-WARSH(30)	11,631	9,706 (83%)	368	188 (51%)	186 (51%)	21	3	2	1	-	
INTERPOL(8)	5,110	1,527 (30%)	43	41 (95%)	37 (86%)	5	-	2	-	-	
MANNERS(128)	849	561 (66%)	328	322 (98%)	317 (97%)	8	-	-	1	-	
NSP_GRND(12)	547	169 (31%)	10	6 (60%)	5 (50%)	3	1	1	-	-	
NSP_GRND(36)	81,835	10,683 (13%)	1,434	502 (35%)	494 (34%)	3	1	1	-	-	
SUM(1000,100)	6,773	3,488 (51%)	215	135 (63%)	<i>N/A</i>	4	-	1	-	-	
TURING(20)	10,372	7,387 (71%)	761	280 (37%)	276 (36%)	11	1	4	1	5	
WFS(200)	2,489	2,143 (86%)	71	67 (94%)	67 (94%)	44	-	4	-	-	
AVERAGE		51.1%		71.5%	56.2%						

Table 10.2: Benchmark results (in average ms) for non-reactive CHR rules.

	SWI		JCHR		total # rules	n-headed prop.				
	history	idempotence	history	idemp.		1	2	3	>	
EQ(35)	3,465	1,931 (56%)	47	19 (40%)	4	-	1	-	-	
LEQ(70)	3,806	1,236 (32%)	85	35 (41%)	4	-	1	-	-	
INTERVAL(21)	22,622	17,611 (78%)	8	5 (62%)	25	1	4	5	2	
INTERVAL(42)	<i>timeout</i>	<i>timeout</i>	54	28 (52%)	25	1	4	5	2	
MINMAX(15)	4,826	3,631 (75%)	133	82 (63%)	55	2	4	-	-	
NSP(12)	1,454	1,036 (71%)	12	8 (67%)	3	1	1	-	-	
NSP(36)	<i>timeout</i>	<i>timeout</i>	1,434	621 (43%)	3	1	1	-	-	
NSP_GRND(12)	547	164 (30%)	10	6 (60%)	3	1	1	-	-	
NSP_GRND(36)	81,835	10,485 (13%)	1,365	496 (36%)	3	1	1	-	-	
TIMEPOINT(16)	1,684	1,312 (78%)	404	317 (78%)	7	-	2	-	-	
AVERAGE		54.1%		54.2%						

Table 10.3: Benchmark results (in average ms) for idempotent CHR rules.

columns contain the reference timings (in milliseconds) when using a propagation history, and the last five columns provide an overview of the number of rules and propagation rules in the different programs.

The *non-react* columns in Table 10.2 contain the results when the optimisations of Section 10.2 are used. For the ‘non-react+’ measurements, loop-invariant code motion was applied to the identifier comparisons (cf. Section 10.2.3; currently only implemented in JCHR⁷). If the history was eliminated using the optimisation of Section 10.2.2, code motion is not applicable (*N/A*). Table 10.3 shows the results for the idempotence-based history elimination of Section 10.3.

Significant performance gains were measured. The benchmarks run about

⁷In JCHR, after code motion, identifier comparisons are integrated in the constraint iterators themselves. These iterators moreover exploit the fact that the stored constraints are often sorted on their identifiers. This can further improve performance.

two times faster on average, and scale better as well. For SWI, we even expect considerable better timings once the identifier comparisons are integrated into the loop-invariant code motion optimisation. Our numbers may also suggest a hash-based history (JCHR) is better suited than a tree-based one (SWI).

The space complexity of the propagation histories has become optimal as well. Unoptimised, the worst-case space consumption of a propagation history, $\mathcal{O}(|\mathbb{S}|^n)$, is linear in the number of possible rule applications. Using our optimisations, propagation histories consume no space at all. This may even improve the space complexity of the entire CHR program.

10.5 Conclusions

While there is a vast literature on CHR compilation and optimisation, propagation histories never received much attention. Maintaining a propagation history, however, comes at a considerable runtime cost, both in time and in space, even when applying the optimisations we introduced in Section 10.1. In this work, we resolved this apparent discrepancy, and introduced several innovative optimisation techniques that circumvent the maintenance of a history for the majority of CHR propagation rules:

- We introduced the notion of *non-reactive* CHR constraints and rules, and showed that for non-reactive CHR propagation rules very cheap constraint identifier comparisons can be used to prevent reapplication. These comparisons can moreover be moved early in the generated nested loops, thus pruning the search space of possible partner constraints. An optimisation that is even applicable for non-propagation rules. We also identified the class of non-reactive rules for which the history can be eliminated entirely.
- While rules in general-purpose CHR programs are mostly non-reactive, CHR handlers that specify a constraint solver typically are not. We therefore introduced the concept of *idempotence*, and found that most rules in the latter handlers are idempotent. We presented a sufficient syntactic condition for the idempotence of a CHR constraint, and showed that if a propagation rule is idempotent, it may safely be applied more than once matching the same combination of constraints. Interestingly, reapplication is mostly cheaper than maintaining and checking a history.

We proved the correctness of all our optimisations and analyses in the formal framework of CHR's refined operational semantics ω_r , and implemented them in two state-of-the-art CHR systems. Our experimental results show significant performance gains (almost 50% on average) for all known benchmarks that extensively use constraint propagation. Moreover, for most CHR programs, the worst-case space complexity is reduced from $\mathcal{O}(|\mathbb{S}|^n)$ to $\mathcal{O}(|\mathbb{S}|)$.

10.5.1 Related work

Section 10.2.2 can be seen as an extension and formalisation of an optimisation briefly presented by Duck (2005). This ad-hoc optimisation was restricted to fixed CHR constraints, and lacked a formal correctness proof. Besides this, no real work on propagation history optimisation existed prior to ours. As discussed next though, there have been some proposals to abandon the history, typically accompanied by the introduction of set semantics or idempotent constraints.

Since the propagation history contributes to significant performance issues when implementing CHR in a tabling environment (see e.g. Schrijvers and Warren 2004), Sarna-Starosta and Ramakrishnan (2007) propose an alternative set-based CHR semantics, and argues that it does not need a propagation history. Our results, however, show that abandoning CHR's familiar multiset-based semantics is not necessary: indeed, our optimisations eliminate the history-related performance issues whilst preserving the ω_r -semantics.

More recently, Betz, Raiser, and Frühwirth (2009) proposed ω_l as a more declarative alternative for ω_t . Unlike ω_t , ω_l has no propagation history. They introduce a new type of CHR constraints, called *persistent constraints*. These are the equivalent of replicated ('banged') resources in linear logic, or replicated processes $!P$ in process calculi. Persistent constraints are (theoretically) never removed from the store, and are therefore inherently idempotent. It is not yet clear what the impact this new, more elegant semantics will have though.

Lastly, in the context of production rules (cf. Chapter 2), we recently re-discovered that the LEAPS algorithm uses timestamp comparisons to prevent reapplication, analogously to our approach in Section 10.2.3. We discuss this further in the upcoming future work section.

10.5.2 Future work

All formal results in this chapter apply to ω_r -based CHR systems only. In future work, we will extend these results to more general CHR $\mathcal{2}$ programs. Extending idempotence is simply a matter of adjusting the analysis used to determine rule idempotence. How to (best) extend our identifier-based reapplication prevention techniques to priorities and negation, however, requires further research.

Priorities

For ω_r -based systems, we proved that identifier- and history-based reapplication prevention are completely equivalent. The semantics of current priority-based formalisms CHR^{IP} and CHR $\mathcal{2}$, however, are far less deterministic than ω_r . Due to the schedule and late indexing optimisations (Section 8.3.3.1), constraints are no longer always activated in increasing order of identifiers. Consequently, while identifier comparisons remains possible, it will affect the rule application order.

That is: certain rule applications will be stopped, not because they have already fired, but because they will fire in the future.

Negation

More problematic though are rules that contain negation as absence. Firstly, for a correct implementation of their fire-many semantics (cf. Section 6.3), the ad-hoc garbage collection techniques of Section 10.1 that are used by current systems are no longer sufficient. Propagation history tuples of rule instances that become inapplicable because constraints matching a negated condition are added, must be removed eagerly. This is a relatively expensive operation, but seems unavoidable in general. To explain why, we must first discuss the algorithm used by LEAPS (Miranker et al. 1990; Brant 1993; Batory et al. 1994). For simplicity, we use CHR terminology (cf. Table 2.1).

Similar to the positive case, LEAPS assigns a timestamp to each removed constraint to represent the time of its removal. For an active removed constraint, a rule instance is again only valid if its removal timestamp is larger than any other timestamp of matched head conditions. New is that timestamps of negated heads—for which matching removed constraints now of course may also have observed the same rule instance before—are always taken into account as well. For this, a so-called *shadow store* is maintained that contains (in principle) all removed constraints. When some active constraint, added or removed, checks a negation condition (cf. Section 8.2.3), this shadow store is now also checked, and if some shadowed removed constraint exists with a timestamp larger than that of the active constraint, the rule does not fire.

There are two serious problems with this approach though:

1. Firstly, the LEAPS algorithm only considered OPS5 rules, that is, all negated conditions consist only consist of a single (possibly guarded) occurrence (cf. Section 2.2.2). We believe the shadow store approach cannot be (efficiently, if at all) extended to more complex negated conjunctions that consist of a join of multiple constraint occurrences.
2. Secondly, a shadow memory suffers from exactly the same garbage collection issues as a propagation history. Keeping track of all removed constraints indefinitely is clearly not an option. It can be hard to determine though when removed facts are no longer needed, perhaps even harder than for propagation history tuples. And even if redundant shadow constraints are optimally reclaimed, Brant (1993) shows the shadow store still has the exact same worst-case space complexity as a history, that is $\mathcal{O}(|S|^n)$.

For fire-many rules, in general, a history with an eagerly implemented `cleanHistory()` operation may therefore be the only solution. For single-occurrence negated conditions, further experimentation is required to determine which approach is best: keeping a propagation history or a shadow store?

Chapter 11

Conclusions

Ideas are like rabbits. You get a couple and learn how to handle them, and pretty soon you have a dozen.

— **John Steinbeck** (1902–1968)
American writer

11.1 Contributions

Our goal was the design and implementation of a more practically usable, declarative CHR system. We now briefly summarise how each chapter contributed to this goal.

11.1.1 CHR language design

In Chapter 5, we proposed CHR2 as a solid basis for a next generation of more declarative and practical CHR systems. CHR2 offers a more elegant, streamlined syntax, and allows the program’s logic to be specified using truly declarative rules. Among other things, CHR2 also offers convenient syntax to declare constraint invariants, a feature that has proven invaluable for the optimising compilation in Part III. We formally specified and studied the operational semantics of the CHR2 language, and designed it to be as non-deterministic as possible while still effectively controllable. Unlike in the currently prevalent ω_r -based systems, the CHR2 system itself normally fully decides the execution strategy. If needed, rule application can be controlled using a combination of two orthogonal, familiar control mechanisms: selective sequentiality (similar to existing ω_r -based systems) and priority constraints (further improving on the work of De Koninck (2008)).

Next, in Chapter 6, we further extend CHR2 with aggregates such as negation as absence, `sum`, `min`, and `findall`. By eliminating the need for cumbersome,

cross-cutting, and error-prone auxiliary constructs, these powerful language extensions improve CHR's expressiveness and conciseness considerably, as shown by a number of case studies. All pragmatic advantages of declarative programming are thus regained. We have fully explored the language design space for aggregate conditions, and worked out a very general framework that supports e.g. user-defined and nested aggregates. We moreover outlined why CHR2 is far better suited to host aggregates than traditional CHR variants (that is, compared e.g. to the implementation for ω_r -based CHR systems we presented in Van Weert et al. 2008).

In the final chapter of Part II, we presented our general design philosophy for natural, practical embeddings of CHR in imperative host languages. We highlighted the design challenges faced, and motivated and discussed our design decisions. Unlike other approaches, our focus has been on the seamless integration and interaction with the imperative host. As a substantial proof-of-concept case study, we implemented the K.U.Leuven JCHR system. The result is a user-friendly, flexible, and highly efficient CHR system for Java, that has already been used in at least two industrial applications. Its successor, JCHR2, is also a first reference implementation of a substantial subset of CHR2.

11.1.2 Optimising implementation of CHR

In Chapter 8, we port the classic CHR(Prolog) compilation scheme to an imperative setting, and present numerous existing and new CHR optimisations in one coherent framework. We extended the basic evaluation methodology and optimisations to the more expressive and less deterministic CHR2 language, extended with negation as absence. We introduced many new optimisations and techniques developed for the different K.U.Leuven JCHR systems, and discussed how we have significantly improved or refined existing ones. The resulting JCHR systems are shown to consistently outperform other CHR systems and mainstream production rule systems, typically by several orders of magnitude.

The last two chapters treat two important compiler optimisations in greater detail. In Chapter 9, we demonstrated the technical problems we experienced with recursive CHR programs—and any non-trivial CHR program *is* recursive—when using the traditional compilation scheme. Particularly when compiling to an imperative target language, executing recursive programs frequently resulted in call stack overflows. We therefore redesigned the compilation scheme. Using our improved techniques, recursive CHR programs now scale properly, also for imperative host languages. Tail recursive CHR programs even execute in optimal constant stack space, independent of the host language.

Chapter 10 deals with reapplication prevention of CHR propagation rules. We found that the traditional approach of maintaining a propagation history is overly costly, both in space and in time. We therefore developed two novel

optimisation techniques to improve the performance of propagation rules. Firstly, for non-reactive CHR rules—most rules in general-purpose CHR programs are non-reactive—reapplication can be prevented far more efficiently by simply comparing the timestamps of the CHR constraints involved in the rule application. Secondly, for idempotent CHR rules—which most rules in constraint solvers are—we showed that allowing reapplication is not only allowed, but, more interestingly, mostly cheaper than preventing it as well. All optimisations are formally proven correct in the ω_r framework. Together, our two optimisations ensure that for the majority of CHR propagation rules reapplication prevention no longer consumes space. Empirical evaluation shows that propagation-intensive benchmarks run about twice as fast.

11.2 Future Work

Much progress has been made by CHR research in the past decade. However, a few difficult questions are still unresolved, and in the meantime many more problems have become apparent. In our view the following three topics are *grand challenges* that must be addressed by the CHR research community in the next decade. These three grand challenges are not only of technical interest, they are also vital for the further adoption of the CHR community and user-base.

- 1. Programming environments and tools.** If measured by current standards, which dictate that a language is only as good as its tools, CHR is a poor language indeed. Modern IDE support for CHR is virtually non-existent (sole exception is the rudimentary JCHRIDE tool by Abdennadher and Fawzy (2008)), and CHR’s (ad-hoc) debugging tools are still very immature. While several strong theoretical results have been obtained in the field of program analysis for CHR, little (if any) effort has been made to embody these results into a practical tool for day-to-day programming. For example, programmers have to manually check for confluence, and, in the case of non-confluence, complete their solvers by hand. There is a great opportunity for taking these theoretical results and establishing their practical relevance by creating the right tools.
- 2. Parallelism, concurrency.** Recent theoretical work (Frühwirth 2005b; Meister 2006) confirms CHR’s inherent aptness for parallel programming. Truly leveraging the full power of current and future multi-core processors through CHR, however, requires practical, efficient, concurrent implementations. Currently, these implementations are still in early stages (cf. Section 4.5 for a discussion of some early Haskell-based prototypes). Many important problems are still to be researched in this domain, from language features and semantics, to analysis, implementation, and optimisation.

3. Scaling to real-life programs. Strong theoretical results have been obtained concerning the performance of CHR, and these have also been reflected in the actual runtimes of CHR programs. However, CHR is still at least one or two orders of magnitude slower than most conventional programming languages and constraint solvers. This becomes particularly apparent for CHR applications that surpass the toy research programs of 10 lines: industrial applications for instance, such as those mentioned in Section 4.6.4, easily count 100 to 1,000 lines or more. Our compilation scheme is insufficiently capable of handling such programs. Some potential scalability aspects are:

- huge constraint stores that have to be persistent and/or distributed;
- dynamic optimisations and JIT compilation, as discussed in Section 8.6;
- incremental compilation, run-time rule assertion;
- reflection, higher-order / meta-programming

These and other aspects must be investigated to achieve further industrial adoption.

Appendices

Appendix A

Heuristics for an A[★] Join Ordering Algorithm

Based on their generic cost formula for CHR join ordering, De Koninck (2008, Chapter 6) and Sneyers (2008, Chapter 9) propose a heuristic for the use in an A[★]-based join ordering implementation. In this appendix, we show first that this heuristical underestimate is incorrect, and then derive a correct, equally efficiently computable heuristic. We only consider regular CHR heads here. The extension towards more expressive head conditions is not hard.

We assume the reader is familiar with the A[★] algorithm (Hart et al. 1972); join ordering was introduced in Section 8.3.2.7. Briefly, A[★] is used to implement (nested-loop) join ordering as follows. The algorithm maintains a pool of partial joins (initially a single, empty join). In each iteration, the most promising partial join is heuristically selected, and gives rise to n new partial joins, one for each remaining join partner. To improve the runtime complexity from $\mathcal{O}(n!)$ to $\mathcal{O}(n \cdot 2^n)$, a closed set—a standard A[★] optimisation—of already expanded joins is used, where different permutations of the same set of join partners is treated as identical. A detailed discussion is outside the scope of this appendix.

For a given set of remaining, not-yet-joined occurrences, the problem at hand is thus to find a lower bound on the estimated cost of computing the remainder of the join, with ‘cost’ defined by the cost formula of De Koninck and Sneyers (2007). This heuristic must be *admissible*, that is, it may never exceed the actual remaining cost estimate given by the cost formula.

For a detailed description of the cost function, including full formal definitions, underlying assumptions, etc., we refer to (De Koninck 2008; Sneyers 2008). For brevity, we only give incomplete, informal descriptions. We distinguish two types

of guards: *a-priori guards*—tested using indexes—and *a-posteriori guards*—the remaining guards. For simplicity, we assume only simple equality guards are tested a priori.¹ Our heuristics are based on the following concepts:

- The *minimal multiplicity* μ^{min} of an occurrence is heuristically estimated as the expected number of constraints that satisfy the (implicit) a-priori equality guards on the occurrence’s arguments, assuming all shared variables are given (or in other words: assuming it is looked up as the last partner in the join order, using optimal equality indexing). If a set semantics functional dependency exists, for instance, μ^{min} is equal to 1.
- The *maximal (a-posteriori) selectivity* σ_*^{max} of an occurrence is heuristically estimated as the expected probability that the a-posteriori guards hold for a given constraint matching the a-priori guards, again assuming all these guards can be tested. Note that the *maximal* selectivity is actually the *minimal* probability of entailment.
- We further define $\gamma^{min} = \mu^{min} \cdot \sigma_*^{max}$ for each occurrence, intuitively the *minimal cardinality* of the set of constraints matching that occurrence.

Important is that, for each occurrence, these values only have to be estimated once. We refer to the work of De Koninck and Sneyers for a detailed description on how to estimate individual multiplicities and selectivities.

Suppose a partial join has length n , and a set X of partners yet has to be joined. Let Θ be any join order starting with the n already fixed partners. Then the actual cost formula that determines the cost of joining the remaining partners X in that join order is of the form:²

$$Cost(X) = |\mathcal{J}_\Theta^n| \cdot \sum_{i=1}^{|X|} \left(\left(\prod_{j=1}^{i-1} \mu^\Theta(n+j) \cdot \sigma_*^\Theta(n+j) \right) \cdot \mu^\Theta(n+i) \right) \quad (\text{A.1})$$

with $|\mathcal{J}_\Theta^n|$ the size of the partial join (cf. De Koninck and Sneyers 2007), and $\mu^\Theta(i)$ and $\sigma_*^\Theta(i)$ the actual multiplicities and selectivities of joining the i th partner of Θ . Because $|\mathcal{J}_\Theta^n|$ depends only on the fixed partial join, the goal of good heuristic H is to be a tight lower bound on the remaining sum.

Original heuristic

Let C_X be the sequence of γ^{min} values of the occurrences in X , sorted from small to large, and M_X^0 and S_X^0 the sequences of μ^{min} and σ_*^{max} values of the

¹We also estimate the a-priori selectivity $\sigma_{eq} = 1$ (cf. De Koninck and Sneyers 2007).

²We have reordered the original cost formula considerably. It is trivial though to verify that our version is equivalent to that of De Koninck (2008) and Sneyers (2008).

corresponding heads, that is: $\forall i : C_X[i] = M_X^0[i] \cdot S_X^0[i]$. To compute the C_X , M_X^0 , and S_X^0 sequences, it suffices to sort all occurrences of a given head once.

Using this notation, the heuristic of De Koninck and Sneyers is given by:

$$H_0(X) = \sum_{i=1}^{|X|} \left(\prod_{j=1}^{i-1} C_X[j] \right) \cdot M_X^0[i] \quad (\text{A.2})$$

Unfortunately, this heuristic is inadmissible. The premise of this heuristic is that, by sorting the γ^{min} values, the sum in (A.2) is minimized. To show that this premise does not hold, we swap the elements α and β of sequences C , M_0 and S_0 ($1 \leq \alpha < \beta \leq |X|$), thus obtaining the sum:

$$H'_0(X) = \sum_{i=1}^{|X|} \left(\prod_{j=1}^{i-1} C'_X[j] \right) \cdot M_X^0[i]$$

Clearly, the terms for $i < \alpha$ and $i > \beta$ remain unchanged after swapping, and

$$\begin{aligned} H_0(X) - H'_0(X) = & \left(\prod_{j=1}^{\alpha-1} C_X[j] \right) \cdot \left(M_X^0[\alpha] - M_X^0[\beta] \right) \\ & + (C_X[\alpha] - C_X[\beta]) \cdot (M_X^0[\alpha+1] + \dots) \\ & + (S_X^0[\alpha] - S_X^0[\beta]) \cdot M_X^0[\alpha] \cdot M_X^0[\beta] \cdot \prod_{k=\alpha+1}^{\beta-1} C_X[k] \end{aligned}$$

If the heuristics' premise were correct, then $H_0(X) \leq H'_0(X)$. But then not only must $C_X[\alpha] \leq C_X[\beta]$, but also $M_X^0[\alpha] \leq M_X^0[\beta]$ and $S_X^0[\alpha] \leq S_X^0[\beta]$. In general, however, sorting their products does not guarantee that sequences M^0 and S^0 are sorted. A counter-example is easily obtained by $C = [1, 3]$, $M^0 = [2, 15]$ and $S^0 = [0.5, 0.2]$, where the latter is not sorted.

Correct heuristics

A first correct underestimate is derived as follows. Observe that the i th term in the sum of the actual cost (A.1) is given by a product of i σ_\star^\ominus and $i+1$ μ^\ominus values. A correct lower bound for the i th term is thus the product of the i smallest σ_\star^{max} , and the $i+1$ smallest μ^{min} values. First, we therefore sort both the σ_\star^{max} and the μ^{min} values of all occurrences in X in two sequences S_X and M_X . Again, in practice, two global S and M lists are computed, from which S_X and M_X are readily derived. The heuristic H_1 is given by:

$$H_1(X) = \sum_{i=1}^{|X|} \left(\prod_{j=1}^{i-1} M_X[j] \cdot S_X[j] \right) \cdot M_X[i]$$

This heuristic only coincides with H_0 if the sequences M_X^0 and S_X^0 happen to be sorted, which as shown earlier is not always the case.

In H_1 , we observe that $M[i]$ and $S[i]$ generally do not originate from the same occurrence, while in the actual cost, the $\mu^\ominus(i) \cdot \sigma_\star^\ominus(i)$ factors do belong to the same occurrence. An alternative underestimate is thus obtained as follows. We again use a sequence C_X defined as before, and the smallest minimal multiplicity of all occurrences in X , i.e. $M_X[1]$. Then

$$H_2(X) = \sum_{i=1}^{|X|} \left(\prod_{j=1}^{i-1} C_X[j] \right) \cdot M_X[1]$$

Again, each term is an obvious underestimate of the corresponding term in the actual cost. The difference with H_0 is that instead of multiplying with $M_X^0(i)$, each term is multiplied with $M_X[1]$, a trivially safe underestimate.

The $M_X[1]$ factor is a potentially very poor underestimate, but there does not seem to be obvious way to improve it. Note that there is no clear winner when comparing H_1 and H_2 . Obviously:

$$H_1(X) = M_X[1] \cdot \sum_{i=1}^{|X|} \left(\prod_{j=1}^{i-1} S_X[j] \cdot M_X[j+1] \right)$$

and so

$$H_1(X) - H_2(X) = M_X[1] \cdot \sum_{i=1}^{|X|} \left(\prod_{j=1}^{i-1} S_X[j] \cdot M_X[j+1] - \prod_{j=1}^{i-1} C_X[j] \right)$$

Although neither of these heuristics is thus superior in itself, we can construct one that is as follows:

$$H_3(X) = M_X[1] \cdot \sum_{i=1}^{|X|} \max \left(\prod_{j=1}^{i-1} S_X[j] \cdot M_X[j+1], \prod_{j=1}^{i-1} C_X[j] \right)$$

This is the heuristic currently used by JCHR2. It provides the best possible underestimates, while still remaining admissible and efficiently computable. We need only to compute three sorted sequences M , S , and C containing the values for all occurrences in the head once. Using these sequences, computing the heuristics then has a reasonable runtime cost linear in the number of remaining partners.

Appendix B

Anti-Monotony-based Delay Avoidance

This appendix introduces the formal changes to the **Solve** transition of the refined operational semantics ω_r required for both anti-monotony-based delay avoidance (Section 8.3.5.2) and the optimisation of non-reactive propagation rules (Section 10.2). It corrects earlier formulations by Schrijvers and Demoen (2004a). In the context of delay avoidance, they proposed the following version of **Solve**:

1. **Solve**[†] $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle S ++ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : \exists V_1, V_2 : vars(c) = V_1 \cup V_2 \wedge V_1 \subseteq fixed(\mathbb{B}) \wedge$ all variables in V_2 appear only in arguments of c that are anti-monotone in \mathcal{P} .

We defined anti-monotonicity is defined in Section 10.2.1 (Definitions 10.2–10.3).

Proposition B.1. *Using the notation we used in Section 10.2, this **Solve**[†] transition is equivalent to:*

1. **Solve**[‡] $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle S ++ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : delay_vars_{\mathcal{P}}(c) \subseteq fixed(\mathbb{B})$.

Proof. Let $c \in \mathbb{S} \setminus S$, with S defined as in **Solve**[†]. Then sets V_1 and V_2 exist, as defined in **Solve**[†]. By definition, $V_2 \subseteq vars(c) \setminus delay_vars_{\mathcal{P}}(c)$, and thus $V_2 \cap delay_vars_{\mathcal{P}}(c) = \emptyset$. Therefore, $delay_vars_{\mathcal{P}}(c) \subseteq V_1 \subseteq fixed(\mathbb{B})$.

Conversely, assume $c \in \mathbb{S} \setminus S$, with S defined as in **Solve**[‡]. Then the required sets V_1 and V_2 exist: $V_1 = delay_vars_{\mathcal{P}}(c)$ and $V_2 = vars(c) \setminus V_1$. \square

Schrijvers and Demoen (2004a) then provide a proof that the resulting semantics is correct with respect to the original refined operational semantics ω_r , where **Solve** is specified as:

1. **Solve**^{*} $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle S ++ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ where b is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\text{vars}(\mathbb{S} \setminus S) \subseteq \text{fixed}(\mathbb{B})$.

That is, all constraints with at least one non-fixed argument have to be reactivated. The original specification of the ω_r semantics therefore explicitly prohibit any form of delay avoidance for non-fixed arguments.

Example B.1. Consider for instance the following CHR program:

```
c(X) ⇒ X = 2, b.
c(_), a ⇔ true.
c(_), b ⇔ true.
```

For the query ‘ $a, c(X)$ ’ with X a free logical variable, **Solve**^{*} specifies that the $c(X)$ constraint has to be reactivated when ‘ $X = 2$ ’ is added to the built-in constraint solver, which leads to a final constraint store $\{b\#3\}$. This is the only final store allowed by the original refined semantics. However, as the program is clearly anti-monotone in c ’s argument, the **Solve**[‡] transition might not reactivate c , which then leads to an incorrect final constraint store $\{a\#1\}$.

This counterexample shows the proof by Schrijvers and Demoen (2004a) must be wrong. The essential problem is that **Solve**^{*} specifies that constraints with non-fixed arguments have to be reactivated, even if the newly added built-in constraint does not enable any new matchings with them. This problem is not restricted to delay avoidance. It was first noted by both Duck (2005) and Schrijvers (2005) in a different context:

Example B.2. Suppose we slightly alter our previous example as follows:

```
c(X) ⇒ Y = 2, b.
c(_), a ⇔ true.
c(_), b ⇔ true.
```

For the query ‘ $a, c(X)$ ’ with X a free logical variable the original **Solve**^{*} transition specifies that the $c(X)$ constraint must be reactivated when ‘ $Y = 2$ ’ is added to the built-in constraint solver. The only final store allowed by the original refined semantics is thus $\{b\#3\}$. However, actual CHR implementations will not reactivate the $c(X)$ constraint, as the newly added ‘ $Y = 2$ ’ constraint does not affect X , the only variable occurring in $c(X)$.

Because the original refined operational semantics is thus inconsistent with the behaviour of actual (Prolog) CHR implementations, a slightly more relaxed version of the **Solve** transition was defined in (Duck 2005; Schrijvers 2005). This is also the version of ω_r we presented in Section 4.2.3. The following theorem shows that our definition of **Solve**^{*} in Section 10.2 is correct with respect to this relaxed ω_r semantics:

Theorem B.2. *Let \mathcal{P} be an arbitrary CHR program, and $\sigma = \langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ an arbitrary state with b a built-in constraint. If $\sigma \rightarrow_{\mathcal{P}} \langle S \uparrow \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ is a valid **Solve**' transition of ω'_r , then it is a valid ω_r **Solve** transition as well.*

Proof. By definition of **Solve**', $S \subseteq \mathbb{S}$, and

- (1) $\forall c \in S : \text{delay_vars}_{\mathcal{P}}(c) \not\subseteq \text{fixed}(\mathbb{B})$,
- (2) $\forall H \subseteq \mathbb{S} : (H = K \uparrow \mathbb{A} \wedge \exists \rho \in \mathcal{P} : \neg \text{appl}(\rho, K, R, \mathbb{B}) \wedge \text{appl}(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$.

As the lower bound of the **Solve** transition in ω_r is also exactly (2), it suffices to prove that $\forall c \in S : \text{vars}(c) \not\subseteq \text{fixed}(\mathbb{B})$. This is obvious given (1), as by definition $\forall c : \text{delay_vars}_{\mathcal{P}}(c) \subseteq \text{vars}(c)$. \square

The optimised semantics by Schrijvers and Demoen (2004a) remains incorrect even with respect to this relaxed ω_r semantics. The reason is that their **Solve**[‡] transition only restricts the constraints that are not reactivated. The constraints that are reactivated, on the other hand, are not restricted:

Example B.3. The following last variant of our example demonstrates this:

$c \Rightarrow \mathbf{x} = 2, b.$
 $c, a \Leftrightarrow \text{true}.$
 $c, b \Leftrightarrow \text{true}.$

For the query ‘ a, c ’ the **Solve** transition of Figure 4.2 specifies that the c constraint may not be reactivated when the ‘ $\mathbf{x} = 2$ ’ constraint is told. This leads to the only final store allowed by the ω_r semantics of Section 4.2.3, namely $\{b\#3\}$. The **Solve**[‡] transition, however, allows the c ’s reactivation. The resulting semantics thus may lead to an incorrect final constraint store $\{a\#1\}$.

The final theorem show that our **Solve**' transition is indeed stronger than **Solve**[‡] (Proposition B.1), since it never reactivates more constraints:

Theorem B.3. *Let \mathcal{P} be a CHR program, and $\sigma = \langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ a state with b a built-in constraint. If $\sigma \rightarrow_{\mathcal{P}} \langle S \uparrow \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ is a valid **Solve**[‡] transition, and $\sigma \rightarrow_{\mathcal{P}} \langle S' \uparrow \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ a valid **Solve**' transition of ω'_r , then $S' \subseteq S$.*

Proof. By definition of **Solve**[‡]: $\forall c \in \mathbb{S} \setminus S : \text{delay_vars}_{\mathcal{P}}(c) \subseteq \text{fixed}(\mathbb{B})$, and by definition of **Solve**': $S' \subseteq \mathbb{S} \wedge \forall c \in S' : \text{delay_vars}_{\mathcal{P}}(c) \not\subseteq \text{fixed}(\mathbb{B})$. Therefore clearly $(\mathbb{S} \setminus S) \cap S' = \emptyset$, and thus $(S' \subseteq \mathbb{S} \wedge S' \cap (\mathbb{S} \setminus S) = \emptyset) \rightarrow S' \subseteq S$. \square

Appendix C

Benchmarks

This appendix contains detailed information on the platforms and benchmark programs used in the main text.

The benchmarks for Table 9.1 were performed on a Intel[®] Core[™] 2 Duo 6400 system with 2 GiB of RAM; all other benchmarks were run on an Intel[®] Pentium[®] 4 CPU 2.80GHz with 1GiB of RAM. In all cases, a Linux operating system was used. Table C.1 lists the software versions used for the different benchmarks.

Table 8.1	YAP 6.0.4, SWI 5.8.3, SICStus 4.1.1, JCHR 1.7, GCC 4.2.4, Eclipse SDK 3.5.1, HotSpot [™] JRE 1.6.0
Table 8.2	CLIPS 6.30 β , Jess 7.1p2, JCHR $\mathcal{2}$ (initial prototype), Eclipse SDK 3.4.2, HotSpot [™] JRE 1.6.0
Table 9.1	SWI 5.6.50, YAP 5.1.2, GCC 4.1.3, JCHR 1.6.0
Tables 9.2–9.3	SWI 5.6.55, JCHR 1.6.1, JDK 1.6.0, HotSpot JRE 1.6.0
Tables 10.1–10.3	SWI 5.6.55 (modified), JCHR 1.6.0 (modified), JDK 1.6.0, HotSpot JRE 1.6.0

Table C.1: Software versions used for benchmarking.

The benchmark programs are mostly either standard CHR or production rule benchmarks, or benchmarks created from standard CHR handlers found on the CHR Website (2010). Table C.2 describes all benchmarks used.

For several shortest-path benchmarks pseudo-random sparse graphs of $\mathcal{O}(N)$ nodes and edges are generated using a graph generator created by Sneyers et al. (2006a, 2009). These graphs consist of a Hamiltonian cycle of N edges with weight 1 from node i to node $i + 1$ (and node N to node 1), and $3N$ random weight edges, 3 from every node to some randomly chosen other.

Benchmark	Description, origin, author, ...
BEER(N)	Produce the lyrics of the well-known ‘ N Bottles of Beer’ song using CHR (without system IO). The original program written by Jon Sneyers appears on http://99-bottles-of-beer.net/ .
BOOL(N)	Classic CHR benchmark from (Schrijvers 2008): performs N -digit binary addition, using an encoding as ternary constraints over booleans
DIJKSTRA(N)	Using Dijkstra’s algorithm to find the shortest path in a sparse graph with $\mathcal{O}(N)$ nodes and edges. Sneyers, Schrijvers, and Demoen (2006a) describe this benchmark in detail.
EQ(N)	Classical CHR benchmark that solves a circular list of N equivalence constraints (cf. Example 5.13).
FIB(N)	Top-down computation of the N first Fibonacci numbers with tabling (origin: Frühwirth 2005c; see also Example 10.3).
FIBBO(N)	Bottom-up computation of the N first Fibonacci numbers (origin: Frühwirth 2005c; cf. Example 10.5).
FLOYD-WARSH(N)	Finding the shortest path between all pairs of nodes of a sparse graph of $\mathcal{O}(N)$ nodes and edges using the Floyd-Warshall algorithm.
GCD(N)	Classic CHR benchmark (Schrijvers 2008) that computes the greatest common divisor of N and 2 using Euclid’s algorithm.
INTERPOL(N)	Linear interpolation of some points up to depth N . Author: Paolo Pilozzi.
INTERVAL(N)	Using T. Frühwirth’s interval domain solver (CHR Website 2010), to solve a sequence of $\mathcal{O}(N)$ addition constraints over N variables.

continued on the next page...

Benchmark	Description, origin, author, . . .
LEQ(N)	Classic CHR benchmark that solves a circular list of N less-or-equal constraints (cf. Example 4.3).
MANNERS(N)	Port of the classic production rules benchmark Miss Manners, a constraint optimisation problem that finds an optimal seating arrangement for N dinner party guests under certain given constraints. Part of the standard “Texas benchmark suite” by Miranker et al. (1991) (see also Illation TM 2007).
MERGESORT(N)	Sorts N integer numbers using mergesort (program author: Thom Frühwirth; origin: CHR Website 2010)
MINMAX(N)	Based on a solver for inequality, minimum and maximum constraints on ground terms, written by T. Frühwirth and C. Holzbaur (CHR Website 2010). The benchmark consists of solving $N - 1$ maximum constraints over N integer variables.
NSP(N)	Finding the shortest path between all pairs of nodes of a sparse graph of $\mathcal{O}(N)$ nodes and edges using a naive shortest path algorithm. Constraint arguments are free logical variables.
NSP_GRND(N)	Variant of the above benchmark where constraint arguments are ground integer values instead.
PRIMES(N)	Classic CHR benchmark (Schrijvers 2008): determine all primes numbers up to N using the Sieve of Eratosthenes.
PRIMES_SWAPPED(N)	Variant of the previous handler, where non-tail recursion is used instead of tail recursion (analogous to Example 9.1).
RAM_FIB(N)	A benchmark by Sneyers et al. (2009): it computes N Fibonacci-like numbers using a CHR-based RAM simulator (addition is replaced by multiplication to avoid arithmetic operations on large numbers).
SUDOKU	A large program consisting of 86 rules that solves Sudoku puzzles, especially designed to benchmark Rete-based production rule systems (see e.g. Illation TM 2007; the program and the benchmark puzzles are available for instance at Friedman-Hill 2010). The benchmark can be run in two modes: one using a directed search (non-stress mode), and one using a naive search (stress mode).
SUM(N, M)	Computes the sum of the balances of the accounts of N clients, where each client has M accounts with pseudo-random balances. See Example 10.4.

continued on the next page. . .

Benchmark	Description, origin, author, . . .
$\text{TAK}(X, Y, Z)$	Computation of Gabriel and McCarthy’s version of the Takeuchi function, an often-used benchmark for testing recursion optimisations.
$\text{TIMEPOINT}(N)$	Solve $1000N$ difference constraints over $1000N$ time points using T. Frühwirth’s and C. Holzbaur’s time point constraint handler (CHR Website 2010).
$\text{TURING}(N)$	Using a Turing machine simulator to execute a Turing program to copy a sequence of N consecutive cells. The simulator was written by Sneyers (2008).
$\text{UNION}(N)$	Performs N disjoint set unions using an optimal union-find algorithm. Benchmark by (Schrijvers and Frühwirth 2006).
$\text{WALTZ}(N)$	Part of the ‘Texas benchmark suite’ (Miranker et al. 1991), so named after the University of Texas at Austin, the affiliation of its authors (Brant, Grose, Lofaso, and Miranker 1991). This standard production rule benchmark interpretes line drawings of some three-dimensional scene using Waltz (1975)’s seminal algorithm (cf. Example 2.1).
$\text{WALTZDB}(N)$	A more general version of the previous benchmark, designed to handle drawings with junctions of four to six lines, while Waltz only does junctions of two or three (Miranker et al. 1991).
$\text{WORDGAME}(N)$	Solves the classic cryptarithmic puzzle ‘GERALD + DONALD = ROBERT’ N times using naive generate-and-test (origin: Friedman-Hill 2010).
$\text{WFS}(N)$	Computes the well-founded semantics of a simple 3-valued logic program (benchmark origin: Schrijvers and Demoen 2004b; Schrijvers 2005).

Table C.2: Descriptions of all benchmarks used in this dissertation.

Bibliography

In this bibliography, the following acronyms for common workshop, conference, book series and journal names are used:

CP — *Intl. Conference on Principles and Practice of Constraint Programming*
ENTCS — *Electronic Notes in Theoretical Computer Science* (published by Elsevier)
ICLP — *International Conference on Logic Programming*
LNAI — *Lecture Notes in Artificial Intelligence* (published by Springer)
LNCS — *Lecture Notes in Computer Science* (published by Springer)
LOPSTR — *Intl. Symposium on Logic-Based Program Synthesis and Transformation*
PPDP — *Intl. ACM SIGPLAN Conf. Principles & Practice of Declarative Programming*
TPLP — *Theory and Practice of Logic Programming* (by Cambridge University Press)
WFLP — *Intl. Workshop on Functional and (Constraint) Logic Programming*
WLP — *Intl. Workshop on (Constraint) Logic Programming*
WLPE — *Intl. Workshop on Logic Programming Environments*

ABDENNADHER, S. 1997. Operational semantics and confluence of constraint propagation rules. In *CP '97* (Schloß Hagenberg, Austria), G. Smolka, Ed. LNCS, vol. 1330. Springer, 252–266.

ABDENNADHER, S. 2000. A language for experimenting with declarative paradigms. In *RCoRP '00(bis): Proc. 2nd Workshop on Rule-Based Constraint Reasoning and Programming* (Singapore), T. Frühwirth et al., Eds.

ABDENNADHER, S. 2001. Rule-based constraint programming: Theory and practice. Habilitationsschrift, Institute of Computer Science, LMU, Munich, Germany.

ABDENNADHER, S. AND CHRISTIANSEN, H. 2000. An experimental CLP platform for integrity constraints and abduction. In *FQAS '00: Proc. 4th Intl. Conf. Flexible Query Answering Systems* (Warsaw, Poland). Springer, 141–152.

ABDENNADHER, S. AND FAWZY, S. 2008. JCHRIDE: An Integrated Development Environment for JCHR. In *WLP '08*, S. Schwarz, Ed. University Halle-Wittenberg, Institute of Computer Science, Technical report 2008/08, Dresden, Germany, 1–6.

ABDENNADHER, S. AND FRÜHWIRTH, T. 1998. On completion of Constraint Handling Rules. In Maher and Puget (1998), 25–39.

- ABDENNADHER, S. AND FRÜHWIRTH, T. 1999. Operational equivalence of CHR programs and constraints. In *CP '99* (Alexandria, Virginia, USA), J. Jaffar, Ed. LNCS, vol. 1713. Springer, 43–57.
- ABDENNADHER, S. AND FRÜHWIRTH, T. 2004. Integration and optimization of rule-based constraint solvers. In *LOPSTR '03* (Uppsala, Sweden), M. Bruynooghe, Ed. LNCS, vol. 3018. Springer, 198–213.
- ABDENNADHER, S., FRÜHWIRTH, T., AND HOLZBAUR, C., Eds. 2005. *Special Issue on Constraint Handling Rules. Theory and Practice of Logic Programming*, vol. 5(4–5). Cambridge University Press.
- ABDENNADHER, S., FRÜHWIRTH, T., AND MEUSS, H. 1999. Confluence and semantics of constraint simplification rules. *Constraints* 4, 2, 133–165.
- ABDENNADHER, S., KRÄMER, E., SAFT, M., AND SCHMAUSS, M. 2002. JACK: A Java Constraint Kit. In *WFLP '01, Selected Papers* (Kiel, Germany), M. Hanus, Ed. ENTCS, vol. 64. Elsevier, 1–17. See also <http://pms.ifi.lmu.de/software/jack/>.
- ABDENNADHER, S. AND MARTE, M. 2000. University course timetabling using Constraint Handling Rules. In Holzbaur and Frühwirth (2000b), 311–325.
- ABDENNADHER, S., OLAMA, A., SALEM, N., AND THABET, A. 2006. ARM: Automatic Rule Miner. In *LOPSTR '06, Revised Selected Papers* (Venice, Italy). LNCS, vol. 4407. Springer.
- ABDENNADHER, S. AND RIGOTTI, C. 2004. Automatic generation of rule-based constraint solvers over finite domains. *ACM TOCL* 5, 2, 177–205.
- ABDENNADHER, S. AND RIGOTTI, C. 2005. Automatic generation of CHR constraint solvers. In Abdennadher, Frühwirth, and Holzbaur (2005), 403–418.
- ABDENNADHER, S. AND SAFT, M. 2001. A visualization tool for Constraint Handling Rules. In *WLPE '01* (Paphos, Cyprus), A. Kusalik, Ed.
- ABDENNADHER, S., SAFT, M., AND WILL, S. 2000. Classroom assignment using constraint logic programming. In *PACLP '00: Proc. 2nd Intl. Conf. and Exhibition on Pract. Appl. of Constraint Techn. and Logic Programming* (Manchester, UK).
- ABDENNADHER, S. AND SCHÜTZ, H. 1998. CHR^V, a flexible query language. In Andreasen, Christiansen, and Larsen (1998), 1–14.
- ABDENNADHER, S. AND SOBHI, I. 2008. Generation of rule-based constraint solvers: Combined approach. In King (2008).
- AGUILAR-SOLIS, D. AND DAHL, V. 2004. Coordination revisited – a Constraint Handling Rule approach. In *IBERAMIA '04: Proc. 9th Ibero-American Conf. on AI* (Puebla, Mexico). LNCS, vol. 3315. 315–324.
- ALBERTI, M., CHESANI, F., GAVANELLI, M., AND LAMMA, E. 2005. The CHR-based implementation of a system for generation and confirmation of hypotheses. In Wolf, Frühwirth, and Meister (2005), 111–122.
- ALBERTI, M., DAOLIO, D., TORRONI, P., GAVANELLI, M., LAMMA, E., AND MELLO, P. 2004. Specification and verification of agent interaction protocols in a logic-based system. In *SAC '04: Proc. 19th ACM Symp. Applied Computing* (Nicosia, Cyprus), H. Haddad et al., Eds. ACM Press, 72–78.

- ALBERTI, M., GAVANELLI, M., LAMMA, E., CHESANI, F., MELLO, P., AND TORRONI, P. 2006. Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence* 20, 2–4, 133–157.
- ALBERTI, M., GAVANELLI, M., LAMMA, E., MELLO, P., AND MILANO, M. 2005. A CHR-based implementation of known arc-consistency. In Abdennadher, Frühwirth, and Holzbaur (2005), 419–440.
- ALBERTI, M., GAVANELLI, M., LAMMA, E., MELLO, P., AND TORRONI, P. 2003. Specification and verification of agent interaction using social integrity constraints. In *LCMAS'03: Logic and Communication in Multi-Agent Systems* (Eindhoven, the Netherlands). ENTCS, vol. 85(2). Elsevier, 94–116.
- ANDREASEN, T., CHRISTIANSEN, H., AND LARSEN, H., Eds. 1998. *FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems* (Roskilde, Denmark). LNAI, vol. 1495. Springer.
- APT, K., KAKAS, A., MONFROY, E., AND ROSSI, F., Eds. 2000. *New Trends in Constraints, Joint ERCIM/Compulog Net Workshop, October 1999, Selected papers* (Paphos, Cyprus). LNCS, vol. 1865. Springer.
- APT, K. R. AND BOL, R. 1994. Logic Programming and negation: A survey. *Journal of Logic Programming* 19, 9–71.
- APT, K. R. AND MONFROY, E. 2001. Constraint programming viewed as rule-based programming. *TPLP* 1, 6, 713–750.
- BAADER, F. AND NIPKOW, T. 2003. *Term Rewriting and all that*. Cambridge University Press.
- BACHANT, J. AND MCDERMOTT, J. 1984. R1 revisited: Four years in the trenches. *AI Magazine* 5, 3.
- BADEA, L., TILIVEA, D., AND HOTARAN, A. 2004. Semantic Web Reasoning for Ontology-Based Integration of Resources. *PPSWR '04: Proc. 2nd Intl. Workshop on Principles And Practice Of Semantic Web Reasoning* 3208, 61–75.
- BAKER, H. G. 1995. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *SIGPLAN Notices* 30, 9, 17–20.
- BALI, M. 2009. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing.
- BARRANCO-MENDOZA, A. 2005. Stochastic and heuristic modelling for analysis of the growth of pre-invasive lesions and for a multidisciplinary approach to early cancer diagnosis. Ph.D. thesis, Simon Fraser University, Burnaby, Canada.
- BARTÁK, R. 1999. Constraint Programming: In pursuit of the Holy Grail. In *Proc. Week of Doctoral Students (WDS'99)* (Prague, Czech Republic). MatFyzPress, 555–564.
- BATORY, D. 1994. The LEAPS algorithms. Tech. Rep. CS-TR-94-28, University of Texas.
- BATORY, D., THOMAS, J., AND SIRKIN, M. 1994. Reengineering a complex application using a scalable data structure compiler. *SIGSOFT Softw. Eng. Notes* 19, 5, 111–120.
- BAUER, A. 2003. Compilation of functional programming languages using GCC—Tail calls. M.S. thesis, Institut für Informatik, Technische Univ. München.

- BAVARIAN, M. AND DAHL, V. 2006. Constraint based methods for biological sequence analysis. *J. Universal Comp. Science* 12, 11, 1500–1520.
- BAYARDO, JR., R. J. AND MIRANKER, D. P. 1996. Processing queries for first-few answers. In *CIKM '96: Proc. fifth intl. Conf. Information and Knowledge Management*. ACM, New York, NY, USA, 45–52.
- BÈS, G. G. AND DAHL, V. 2003. Balanced parentheses in NL texts: a useful cue in the syntax/semantics interface. In *Proc. Lorraine-Saarland Workshop on Prospects and Advances in the Syntax/Semantics Interface* (Nancy, France). Poster Paper.
- BETZ, H. 2007. Relating coloured Petri nets to Constraint Handling Rules. In Djelloul, Duck, and Sulzmann (2007), 33–47.
- BETZ, H. AND FRÜHWIRTH, T. 2005. A linear-logic semantics for Constraint Handling Rules. In *CP '05* (Sitges, Spain). LNCS, vol. 3709. Springer, 137–151.
- BETZ, H. AND FRÜHWIRTH, T. 2007. A linear-logic semantics for Constraint Handling Rules with disjunction. In Djelloul, Duck, and Sulzmann (2007), 17–31.
- BETZ, H., RAISER, F., AND FRÜHWIRTH, T. 2009. Persistent constraints in Constraint Handling Rules. In *WLP '09* (Potsdam, Germany), A. Wolf and U. Geske, Eds.
- BEZEM, M., KLOP, J.-W., AND DE VRIJER, R. 2003. *Term rewriting systems*. Cambridge University Press.
- BISTARELLI, S., FRÜHWIRTH, T., MARTE, M., AND ROSSI, F. 2004. Soft constraint propagation and solving in Constraint Handling Rules. *Computational Intelligence: Special Issue on Preferences in AI and CP 20*, 2 (May), 287–307.
- BLOCH, J. ET AL. 2010. The Collections framework: API's and developer guides. <http://java.sun.com/javase/6/docs/technotes/guides/collections/>.
- BOESPFLUG, M. 2007. TaiChi:how to check your types with serenity. *The Monad.Reader* 9, 17–31.
- BOOK, R. V. AND OTTO, F. 1993. *String-rewriting systems*. Springer, London, UK.
- BOUAUD, J. AND VOYER, R. 2004. Behavioral match: Embedding production systems and objects. In Pachet (2004).
- BOUISSOU, O. 2004. A CHR library for SiLCC. Diplomarbeit, Technical University of Berlin, Germany.
- BRACHA, G. 2004. *Generics in the Java Programming Language*. Sun Microsystems.
- BRAND, S. 2002. A note on redundant rules in rule-based constraint programming. In *Joint ERCIM/CologNet Intl. Workshop on Constraint Solving and Constraint Logic Programming, Selected papers* (Cork, Ireland). LNCS, vol. 2627. Springer, 279–336.
- BRAND, S. AND MONFROY, E. 2003. Deductive generation of constraint propagation rules. In *RULE '03: 4th Intl. Workshop on Rule-Based Programming* (Valencia, Spain), G. Vidal, Ed. ENTCS, vol. 86(2). Elsevier, 45–60.
- BRANT, D. A. 1993. Inferencing on large data sets. Ph.D. thesis, University of Texas at Austin.

- BRANT, D. A., GROSE, T., LOFASO, B., AND MIRANKER, D. P. 1991. Effects of database size on rule system performance: Five case studies. In *Proc. 17th Intl. Conf. Very Large Data Bases (VLDB'91)*. Morgan Kaufmann.
- BRATKO, I. 2008. *Prolog programming for Artificial Intelligence*, Fourth ed. Pearson Education Canada.
- BRESSAN, S. AND GOH, C. H. 1998. Answering queries in context. In Andreassen, Christiansen, and Larsen (1998), 68–82.
- BREWKA, G. AND EITER, T. 1999. Preferred answer sets for extended logic programs. *Artif. Intell.* 109, 1–2, 297–356.
- BROCK, M. ET AL. 2010. MVEL. <http://mvel.codehaus.org/>.
- BROWNE, J. C., EMERSON, A., GOUDA, M., MIRANKER, D. P., ET AL. 1994. A new approach to modularity in rule-based programming. In *Proc. 6th Intl. Conf. Tools with AI*. IEEE Comp. Society.
- BROWNE, P. 2009. *JBoss Drools Business Rules*. Packt Publishing.
- BROWNSTON, L., FARRELL, R., KANT, E., AND MARTIN, N. 1985. *Programming expert systems in OPS5: an introduction to rule-based programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- BUCCAFURRI, F., Ed. 2003. *AGP '03: Joint Conf. Declarative Programming APPIA-GULP-PRODE* (Reggio Calabria, Italy).
- BURSTALL, R. M. AND DARLINGTON, J. 1977. A transformation system for developing recursive programs. *Journal of the ACM* 24, 1, 44–67.
- CABEDO, L. M. AND ESCRIG, M. T. 2003. Modeling motion by the integration of topology and time. *J. Universal Comp. Science* 9, 9, 1096–1122.
- CALVERT, C. AND KULKARNI, D. 2009. *Essential LINQ*. Addison-Wesley.
- CHIN, W.-N., CRACIUN, F., KHOO, S.-C., AND POPEEA, C. 2006. A flow-based approach for variant parametric types. *SIGPLAN Not.* 41, 10, 273–290.
- CHIN, W.-N., SULZMANN, M., AND WANG, M. 2003. A type-safe embedding of Constraint Handling Rules into Haskell. Honors thesis, School of Computing, National University of Singapore.
- CHR Website 2010. The Constraint Handling Rules (CHR) programming language. <http://dtai.cs.kuleuven.be/CHR>.
- CHRISTIANSEN, H. 2005. CHR grammars. In Abdennadher, Frühwirth, and Holzbaur (2005), 467–501.
- CHRISTIANSEN, H. AND DAHL, V. 2003. Logic grammars for diagnosis and repair. *Intl. J. Artificial Intelligence Tools* 12, 3, 227–248.
- CHRISTIANSEN, H. AND DAHL, V. 2005a. HYPROLOG: A new logic programming language with assumptions and abduction. In Gabbrielli and Gupta (2005), 159–173.
- CHRISTIANSEN, H. AND DAHL, V. 2005b. Meaning in context. In *CONTEXT '05: Proc. 4th Intl. and Interdisciplinary Conf. Modeling and Using Context* (Paris, France), A. Dey, B. Kokinov, and R. Turner, Eds. LNAI, vol. 3554. Springer, 97–111.

- CHRISTIANSEN, H. AND HAVE, C. T. 2007. From use cases to UML class diagrams using logic grammars and constraints. In *RANLP '07: Proc. Intl. Conf. Recent Adv. Nat. Lang. Processing* (Borovets, Bulgaria). 128–132.
- CHRISTIANSEN, N. A., WESTH, A. B., BASBOUS, J. E., AND CHRISTIANSEN, H. 2006. Constraint handling rules — en undersøgelse af interne lagrings mekanismer. Tech. rep., Roskilde Universitetscenter. In Danish.
- CLARK, K. 1978. Negation as failure. In *Logic and Databases*, H. Gallaire, J. Minker, and J. Nicolas, Eds. Plenum Press.
- CLEMENTS, J. AND FELLEISEN, M. 2004. A tail-recursive machine with stack inspection. *ACM Trans. on Prog. Languages and Systems (TOPLAS)* 26, 6, 1029–1052.
- CLOCKSIN, W. F. AND MELLISH, C. S. 2003. *Programming in Prolog*, Fifth ed. Springer.
- COOPER, T. A. AND WOGGIN, N. 1988. *Rule-Based Programming With OPS5*. Morgan Kaufmann Pub.
- COQUERY, E. AND FAGES, F. 2003. TCLP: A type checker for $CLP(\mathcal{X})$. In *WLPE '03* (Mumbai, India), F. Mesnard and A. Serebrenik, Eds. K.U.Leuven, Dept. Comp. Sc., Technical report CW 371. 17–30.
- COQUERY, E. AND FAGES, F. 2005. A type system for CHR. In Schrijvers and Frühwirth (2005b), 19–33.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2009. *Introduction to Algorithms*, Third ed. MIT Press.
- DA FIGUEIRA FILHO, C. S. AND RAMALHO, G. L. 2000. JEOPS - the Java Embedded Object Production System. In *Advances in Artificial Intelligence — IBERAMIA-SBIA 2000: Proc. Intl. Joint Conf. 7th Ibero-American Conference on AI – 15th Brazilian Symposium on AI*. LNCS, vol. 1952. Springer, Atibaia, SP, Brazil.
- DAHL, V. 2004. An abductive treatment of long distance dependencies in CHR. In *CSLP '04: Proc. First Intl. Workshop on Constraint Solving and Language Processing* (Roskilde, Denmark). LNCS, vol. 3438. Springer, 17–31. Invited Paper.
- DAHL, V. AND BLACHE, P. 2005. Extracting selected phrases through constraint satisfaction. In *Proc. 2nd Intl. Workshop on Constraint Solving and Language Processing*.
- DAHL, V. AND GU, B. 2006. Semantic property grammars for knowledge extraction from biomedical text. In Etalle and Truszczyński (2006), 442–443. Poster Paper.
- DAHL, V. AND GU, B. 2007. A CHRg analysis of ambiguity in biological texts. In *CSLP '07: Proc. 4th Intl. Workshop on Constraints and Language Processing* (Roskilde, Denmark). Extended Abstract.
- DAHL, V. AND NIEMELÄ, I., Eds. 2007. *ICLP '07: Proc. 23rd Intl. Conf. Logic Programming* (Porto, Portugal). LNCS, vol. 4670. Springer.
- DAHL, V. AND VOLL, K. 2004. Concept formation rules: An executable cognitive model of knowledge construction. In *NLUCS '04: Proc. First Intl. Workshop on Natural Language Understanding and Cognitive Sciences* (Porto, Portugal).
- DANIEL, B. AND BOSHERNITSAN, M. 2008. Predicting and explaining automatic testing tool effectiveness. Tech. rep., University of Illinois at Urbana-Champaign.

- DE KONINCK, L. 2008. Execution control for Constraint Handling Rules. Ph.D. thesis, K.U.Leuven, Belgium.
- DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2006a. INCLP(\mathbb{R}) - Interval-based nonlinear constraint logic programming over the reals. In Fink, Tompits, and Woltran (2006), 91–100.
- DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2006b. Search strategies in CHR(Prolog). In Schrijvers and Frühwirth (2006), 109–124.
- DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2007a. The correspondence between the Logical Algorithms language and CHR. In Dahl and Niemelä (2007), 209–223.
- DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2007b. User-definable rule priorities for CHR. In Leuschel and Podelski (2007), 25–36.
- DE KONINCK, L. AND SNEYERS, J. 2007. Join ordering for Constraint Handling Rules. In Djelloul, Duck, and Sulzmann (2007), 107–121.
- DEBRAY, S. K. 1988. Unfold/fold transformations and loop optimization of logic programs. In *PLDI '88: Proc. ACM SIGPLAN 1988 conf. on Progr. Language Design and Implementation*. ACM, New York, NY, USA, 297–307.
- DEMOEN, B. 2002. Dynamic attributes, their hProlog implementation, and a first evaluation. Tech. Rep. CW 350, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium. Oct.
- DEMOEN, B. AND LIFSCHITZ, V., Eds. 2004. *ICLP '04: Proc. 20th Intl. Conf. Logic Programming* (Saint-Malo, France). LNCS, vol. 3132. Springer.
- DJELLOUL, K., DAO, T.-B.-H., AND FRÜHWIRTH, T. 2007. Toward a first-order extension of Prolog's unification using CHR: a CHR first-order constraint solver over finite or infinite trees. In *SAC '07: Proc. 2007 ACM Symp. Applied computing* (Seoul, Korea). ACM Press, 58–64.
- DJELLOUL, K., DUCK, G. J., AND SULZMANN, M., Eds. 2007. *CHR '07: Proc. 4th Workshop on Constraint Handling Rules* (Porto, Portugal).
- DOORENBOS, R. B. 1995. Production matching for large learning systems. Ph.D. thesis, Carnegie Mellon University.
- DUCASSÉ, M. 1999. Opium: an extendable trace analyzer for Prolog. *J. Logic Programming* 39, 1–3, 177–223.
- DUCK, G. J. 2004. HaskellCHR. <http://www.cs.mu.oz.au/~gjd/haskellchr>.
- DUCK, G. J. 2005. Compilation of Constraint Handling Rules. Ph.D. thesis, University of Melbourne, Australia.
- DUCK, G. J. AND SCHRIJVERS, T. 2005. Accurate functional dependency analysis for Constraint Handling Rules. In Schrijvers and Frühwirth (2005b), 109–124.
- DUCK, G. J., STUCKEY, P. J., AND BRAND, S. 2006. ACD term rewriting. In Etalle and Truszczyński (2006), 117–131.
- DUCK, G. J., STUCKEY, P. J., GARCÍA DE LA BANDA, M., AND HOLZBAUR, C. 2003. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP '03* (Uppsala, Sweden). ACM Press, 79–90.

- DUCK, G. J., STUCKEY, P. J., GARCÍA DE LA BANDA, M., AND HOLZBAUR, C. 2004. The refined operational semantics of Constraint Handling Rules. In Demoen and Lifschitz (2004), 90–104.
- DUCK, G. J., STUCKEY, P. J., AND SULZMANN, M. 2007. Observable confluence for Constraint Handling Rules. In Dahl and Niemelä (2007), 224–239.
- EHRIG, H. AND ROZENBERG, G., Eds. 1999. *Handbook of Graph Grammars and Computing by Graph Transformations*. Vol. 1–3. World Scientific.
- ESCRIG, M. T. AND TOLEDO, F. 1998a. A framework based on CLP extended with CHRs for reasoning with qualitative orientation and positional information. *J. Visual Languages and Computing* 9, 1, 81–101.
- ESCRIG, M. T. AND TOLEDO, F. 1998b. *Qualitative Spatial Reasoning: Theory and Practice — Application to Robot Navigation*. IOS Press.
- ETALLE, S. AND TRUSZCZYNSKI, M., Eds. 2006. *ICLP '06: Proc. 22nd Intl. Conf. Logic Programming* (Seattle, Washington). LNCS, vol. 4079. Springer.
- FABRET, F., RÉGNIER, M., AND SIMON, E. 1993. An adaptive algorithm for incremental evaluation of production rules in databases. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 455–466.
- FAGES, F. 1997. Constructive negation by pruning. *The Journal of Logic Programming* 32, 2, 85–118.
- FAGES, F., MÁRIO DE OLIVEIRA RODRIGUES, C., AND MARTINEZ, T. 2008. Modular CHR with *ask* and *tell*. In Schrijvers, Raiser, and Frühwirth (2008), 95–110.
- FINK, M., TOMPITS, H., AND WOLTRAN, S., Eds. 2006. *WLP '06: Proc. 20th Workshop on Logic Programming* (Vienna, Austria). T.U.Wien, Austria, INFSYS Research report 1843-06-02.
- FIRAT, A. 2003. Information integration using contextual knowledge and ontology merging. Ph.D. thesis, MIT Sloan School of Management, Cambridge, USA.
- FORGY, C. L. 1979. On the efficient implementation of production systems. Ph.D. thesis, Carnegie Mellon University.
- FORGY, C. L. 1981. OPS5 user's manual. Tech. Rep. CS-81-135, Carnegie-Mellon University.
- FORGY, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19, 17–37.
- FREE SOFTWARE FOUNDATION. 2010. GCC, the GNU Compiler Collection.
- FREUDER, E. C. 1997. In pursuit of the Holy Grail. *Constraints* 2, 1 (Apr.), 57–61.
- FRIEDMAN-HILL, E. 2003. *Jess in Action*. Manning Publications Co.
- FRIEDMAN-HILL, E. 2010. Jess[®], the rule engine for the Java[™] platform. <http://www.jessrules.com/>.
- FRÜHWIRTH, T. 1992. Constraint simplification rules. Tech. Rep. ECRC-92-18, European Computer-Industry Research Centre, Munchen, Germany. July.

- FRÜHWIRTH, T. 1995. Constraint Handling Rules. In *Constraint Programming: Basic and Trends — Selected Papers of the 22nd Spring School in Theoretical Computer Sciences, May 16–20, 1994* (Châtillon-sur-Seine, France), A. Podelski, Ed. LNCS, vol. 910. Springer, 90–107.
- FRÜHWIRTH, T. 1998. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming 37*, 1–3, 95–138.
- FRÜHWIRTH, T. 2000. Proving termination of constraint solver programs. In Apt, Kakas, Monfroy, and Rossi (2000), 298–317.
- FRÜHWIRTH, T. 2001. On the number of rule applications in constraint programs. In *Declarative Programming - Selected Papers from AGP 2000* (La Habana, Cuba), A. Dovier, M. C. Meo, and A. Omicini, Eds. ENTCS, vol. 48. Elsevier, 147–166.
- FRÜHWIRTH, T. 2002a. As time goes by: Automatic complexity analysis of simplification rules. In *KR '02: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning* (Toulouse, France), D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, Eds. Morgan Kaufmann, 547–557.
- FRÜHWIRTH, T. 2002b. As time goes by II: More automatic complexity analysis of concurrent rule programs. In *QAPL '01: Proc. First Intl. Workshop on Quantitative Aspects of Programming Languages* (Florence, Italy). ENTCS, vol. 59(3). Elsevier.
- FRÜHWIRTH, T. 2005a. Logical rules for a lexicographic order constraint solver. In Schrijvers and Frühwirth (2005b), 79–91.
- FRÜHWIRTH, T. 2005b. Parallelizing union-find in Constraint Handling Rules using confluence. In Gabbrielli and Gupta (2005), 113–127.
- FRÜHWIRTH, T. 2005c. Programming with a Chinese horse. Invited Talk at 11th Intl. Conf., CP 2005, Sitges, Spain. (slides).
- FRÜHWIRTH, T. 2005d. Specialization of concurrent guarded multi-set transformation rules. In *LOPSTR '04* (Verona, Italy). LNCS, vol. 3573. Springer, 133–148.
- FRÜHWIRTH, T. 2006a. Complete propagation rules for lexicographic order constraints over arbitrary domains. In *Recent Advances in Constraints — CSCLP '05: Joint ERCIM/CoLogNET Intl. Workshop on Constraint Solving and CLP, Revised Selected and Invited Papers* (Uppsala, Sweden). LNAI, vol. 3978. Springer.
- FRÜHWIRTH, T. 2006b. Deriving linear-time algorithms from union-find in CHR. In Schrijvers and Frühwirth (2006), 49–60.
- FRÜHWIRTH, T. 2009. *Constraint Handling Rules*. Cambridge University Press.
- FRÜHWIRTH, T. ET AL., Eds. 2000. *RCoRP '00: Proc. 1st Workshop on Rule-Based Constraint Reasoning and Programming* (London, UK).
- FRÜHWIRTH, T. AND ABDENNADHER, S. 2001. The Munich rent advisor: A success for logic programming on the internet. *TPLP 1*, 3, 303–319.
- FRÜHWIRTH, T. AND ABDENNADHER, S. 2003. *Essentials of Constraint Programming*. Springer.
- FRÜHWIRTH, T. AND BRISSET, P. 1995. High-level implementations of Constraint Handling Rules. Tech. Rep. ECRC-95-20, European Computer-Industry Research Centre, Munchen, Germany.

- FRÜHWIRTH, T. AND BRISSET, P. 1998. Optimal placement of base stations in wireless indoor telecommunication. In Maher and Puget (1998), 476–480.
- FRÜHWIRTH, T. AND BRISSET, P. 2000. Placing base stations in wireless indoor communication networks. *IEEE Intell. Systems and Their Applications* 15, 1, 49–53.
- FRÜHWIRTH, T., DI PIERRO, A., AND WIKLICKY, H. 2002. Probabilistic Constraint Handling Rules. In *WFLP '02, Selected Papers* (Grado, Italy), M. Comini and M. Falaschi, Eds. ENTCS, vol. 76. Elsevier.
- FRÜHWIRTH, T. AND HOLZBAUR, C. 2003. Source-to-source transformation for a class of expressive rules. In Buccafurri (2003), 386–397.
- FRÜHWIRTH, T. AND MEISTER, M., Eds. 2004. *CHR '04: 1st Workshop on Constraint Handling Rules: Selected Contributions* (Ulm, Germany).
- FRÜHWIRTH, T. W., HEROLD, A., KÜCHENHOFF, V., PROVOST, T. L., LIM, P., MONFROY, E., AND WALLACE, M. 1992. Constraint Logic Programming - an informal introduction. In *Logic Programming in Action, Proc. Second Intl. Logic Programming Summer School*. 3–35.
- GABBRIELLI, M. AND GUPTA, G., Eds. 2005. *ICLP '05: Proc. 21st Intl. Conf. Logic Programming* (Sitges, Spain). LNCS, vol. 3668. Springer.
- GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.
- GANZ, S. E., FRIEDMAN, D. P., AND WAND, M. 1999. Trampolined style. *SIGPLAN Notices* 34, 4, 18–27. Proc. Intl. Conf. on Functional Programming (ICFP'99).
- GANZINGER, H. AND MCALLESTER, D. A. 2002. Logical algorithms. In Stuckey (2002), 209–223.
- GARAT, D. AND WONSEVER, D. 2002. A constraint parser for contextual rules. In *Proc. 22nd Intl. Conf. of the Chilean Computer Science Society* (Copiapo, Chile). IEEE Computer Society, 234–242.
- GARCÍA DE LA BANDA, M. AND PONTELLI, E., Eds. 2008. *ICLP '08: Proc. 24rd Intl. Conf. Logic Programming* (Udine, Italy). LNCS, vol. 5366. Springer.
- GAVANELLI, M., LAMMA, E., MELLO, P., MILANO, M., AND TORRONI, P. 2003. Interpreting abduction in CLP. In Buccafurri (2003), 25–35.
- GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proc. Fifth Intl. Conf. on Logic Programming*. MIT Press, 1070–1080.
- GEURTS, J., VAN OSSENBRUGGEN, J., AND HARDMAN, L. 2001. Application-specific constraints for multimedia presentation generation. In *MMM '01: Proc. 8th Intl. Conf. on Multimedia Modeling* (Amsterdam, The Netherlands). 247–266.
- GIARRATANO, J. C. AND RILEY, G. 1989. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA.
- GIRARD, J.-Y. 1987. Linear logic. *Theoretical Computer Science* 50, 1, 1–101.
- GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The Java Language Specification*, Third ed. The Java Series. Prentice Hall.

- GOURAUD, S.-D. AND GOTLIEB, A. 2006. Using CHRs to generate functional test cases for the Java card virtual machine. In *PADL '06: Proc. 8th Intl. Symp. Practical Aspects of Declarative Languages* (Charleston, SC, USA), P. Van Hentenryck, Ed. LNCS, vol. 3819. Springer, 1–15.
- HAEMMERLÉ, R. AND FAGES, F. 2007. Abstract critical pairs and confluence of arbitrary binary relations. In *RTA '07: Proc. 18th Intl. Conf. Term Rewriting and Applications* (Paris, France). LNCS, vol. 4533. Springer.
- HAMILTON, G. ET AL. 1997. *JavaBeansTM API specification*, 1.01 ed. Sun Microsystems.
- HANSON, E. AND WIDOM, J. 1993. An overview of Production Rules in database systems. *The Knowledge Engineering Review* 8, 2, 121–143.
- HANSON, E. N. 1992. Rule condition testing and action execution in Ariel. In *Proc. 1992 ACM SIGMOD Intl. Conf. on Management of data*. ACM, New York, NY, USA, 49–58.
- HANSON, E. N., BODAGALA, S., HASAN, M., KULKARNI, G., AND RANGARAJAN, J. 1995. Optimized rule condition testing in Ariel using Gator networks. Tech. Rep. TR-95-027, CISE Department, University of Florida.
- HANUS, M. 2006. Adding Constraint Handling Rules to Curry. In Fink, Tompits, and Woltran (2006), 81–90.
- HARALICK, R. M. AND ELLIOTT, G. L. 1979. Increasing tree search efficiency for constraint satisfaction problems. In *IJCAI'79: Proc. of the 6th Intl. Joint Conf. on Art. Intell.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 356–364.
- HART, P. E., NILSSON, N. J., AND RAPHAEL, B. 1972. Correction to “a formal basis for the heuristic determination of minimum cost paths”. *SIGART Bull.* 37, 28–29.
- HECKSHER, T., NIELSEN, S. T., AND PIGEON, A. 2002. A CHRg model of the ancient Egyptian grammar. Unpublished student project report, Roskilde University, Denmark.
- HILL, P. M. AND WARREN, D. S., Eds. 2009. *ICLP '09: Proc. 25th Intl. Conf. Logic Programming* (Pasadena, USA). LNCS, vol. 5649. Springer.
- HOLZBAUR, C. 1992. Metastructures versus attributed variables in the context of extensible unification. In *Proc. 4th Intl. Symposium on Programming Language Implementation and Logic Programming*. Springer, 260–268.
- HOLZBAUR, C. AND FRÜHWIRTH, T. 1999. Compiling Constraint Handling Rules into Prolog with attributed variables. In *PPDP '99* (Paris, France), G. Nadathur, Ed. LNCS, vol. 1702. Springer, 117–133.
- HOLZBAUR, C. AND FRÜHWIRTH, T. 2000a. A Prolog Constraint Handling Rules compiler and runtime system. In Holzbaur and Frühwirth (2000b), 369–388.
- HOLZBAUR, C. AND FRÜHWIRTH, T., Eds. 2000b. *Special Issue on Constraint Handling Rules*. Journal of Applied Artificial Intelligence, vol. 14(4). Taylor & Francis.
- HOLZBAUR, C., GARCÍA DE LA BANDA, M., STUCKEY, P. J., AND DUCK, G. J. 2005. Optimizing compilation of Constraint Handling Rules in HAL. In Abdennadher, Frühwirth, and Holzbaur (2005), 503–531.
- HOPCROFT, J. E. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. Tech. Rep. STAN-CS-71-190, Stanford University, CA, USA.

- HU, Z. AND TAKEICHI, M. 1997. A calculational framework for parallelization of sequential programs. In *Intl. Symp. Information Systems and Technologies for Network Society*. Fukuoka, Japan, 102–109.
- IBARAKI, T. AND KAMEDA, T. 1984. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.* 9, 3, 482–502.
- ILLATIONTM. 2007. Business rule engine benchmarks. <http://illation.com.au/benchmarks/>.
- ISO. 1995. ISO/IEC 13211:1995: Information technology – Programming languages – Prolog.
- ISO. 2003. ISO/IEC 9075:2008: Information technology – Database languages – SQL.
- JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint Logic Programming. In *POPL '87: Proc. 14th ACM SIGACT-SIGPLAN symp. on Princ. of Progr. Lang.* ACM, New York, NY, USA, 111–119.
- JBOS. 2010. Drools 5 – the business logic integration platform. <http://jboss.org/drools/>.
- KEMP, D. B. AND STUCKEY, P. J. 1991. Semantics of logic programs with aggregates. In *Intl. Symp. Logic Programming*. San Diego, USA, 387–404.
- KING, A., Ed. 2008. *LOPSTR '07: 17th Intl. Symp. Logic-Based Program Synthesis and Transformation, Revised Selected Papers* (Kongens Lyngby, Denmark). LNCS, vol. 4915.
- KOSMATOV, N. 2006a. A constraint solver for sequences and its applications. In *Proc. 2006 ACM Symp. on Applied Computing* (Dijon, France). ACM Press, 404–408.
- KOSMATOV, N. 2006b. Constraint solving for sequences in software validation and verification. In *INAP '05: Proc. 16th Intl. Conf. Applications of Declarative Programming and Knowledge Management* (Fukuoka, Japan). LNCS, vol. 4369. Springer, 25–37.
- KOWALSKI, R. 1979. Algorithm = logic + control. *Commun. ACM* 22, 7, 424–436.
- KRAFT, A. 1984. XCON: An expert configuration system at Digital Equipment Corporation. In *The AI Business – Commercial Uses of Artificial Intelligence*, P. H. Winston and K. A. Prendergast, Eds. MIT Press, Cambridge, MA, 43–49.
- KRÄMER, E. 2001. A generic search engine for a Java Constraint Kit. Diplomarbeit, Institute of Computer Science, LMU, Munich, Germany.
- KRISHNAMURTHY, R., BORAL, H., AND ZANIOLO, C. 1986. Optimization of nonrecursive queries. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 128–137.
- LAM, E. S. AND SULZMANN, M. 2006. Towards agent programming in CHR. In Schrijvers and Frühwirth (2006), 17–31.
- LAM, E. S. AND SULZMANN, M. 2008. Finally, a comparison between Constraint Handling Rules and join-calculus. In Schrijvers, Raiser, and Frühwirth (2008), 51–66.
- LAVRAC, N. AND DZEROSKI, S. 1994. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York.

- LEUSCHEL, M. AND PODELSKI, A., Eds. 2007. *PPDP '07: Proc. 9th Intl. Conf. Princ. Pract. Declarative Programming* (Wrocław, Poland). ACM Press.
- LIFSCHITZ, V. 1996. Foundations of logic programming. In *Principles of knowledge representation*, G. Brewka, Ed. Studies In Logic, Language And Information. Center for the Study of Language and Information, Stanford, CA, USA, 69–127.
- LINDHOLM, T. AND YELLIN, F. 1999. *The JavaTM Virtual Machine Specification*, 2 ed. Prentice Hall.
- LLOYD, J. W. 1987. *Foundations of logic programming*, Second extended ed. Springer.
- LÖTZBEYER, H. AND PRETSCHNER, A. 2000. AUTOFOCUS on constraint logic programming. In *LPSE '00: Proc. Intl. Workshop on (Constraint) Logic Programming and Software Engineering* (London, United Kingdom).
- MAHER, M. J. AND PUGET, J.-F., Eds. 1998. *CP '98: Proc. 4th Intl. Conf. Princ. Pract. Constraint Programming* (Pisa, Italy). LNCS, vol. 1520. Springer.
- MARRIOTT, K. AND STUCKEY, P. J. 1998. *Programming with constraints: an introduction*. MIT Press.
- MCCARTHY, D. AND DAYAL, U. 1989. The architecture of an active database management system. *SIGMOD Record* 18, 2, 215–224.
- MCCARTHY, J. 1963. Situations, actions, and causal laws. Tech. rep., Stanford University.
- MCDERMOTT, J. 1980. R1: An expert in the computer systems domain. *Artificial Intelligence* 19, 1, 39–88. Winner of the 1999 AAAI Classic Paper Award.
- MCDERMOTT, J. 1994. R1 (“xcon”) at age 12: lessons from an elementary school achiever. In *Artificial intelligence in perspective*. MIT Press, 241–247.
- MCDERMOTT, J. AND FORGY, C. L. 1978. Production system conflict resolution strategies. In *Pattern-Directed Inference Systems*. Academic Press.
- MEIJER, E., FOKKINGA, M., AND PATERSON, R. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*. LNCS, vol. 523. Springer.
- MEISTER, M. 2006. Fine-grained parallel implementation of the preflow-push algorithm in CHR. In Fink, Tompits, and Woltran (2006), 172–181.
- MEISTER, M., DJELLOUL, K., AND FRÜHWIRTH, T. 2006. Complexity of a CHR solver for existentially quantified conjunctions of equations over trees. In *CSCLP '06: Proc. 11th Annual ERCIM Workshop on Constraint Solving and Constraint Programming* (Caparica, Portugal), F. Azevedo et al., Eds. LNCS, vol. 4651. Springer, 139–153.
- MEISTER, M., DJELLOUL, K., AND ROBIN, J. 2007. A unified semantics for Constraint Handling Rules in transaction logic. In *LPNMR '07: Proc. 9th Intl. Conf. Logic Programming and Nonmonotonic Reasoning* (Tempe, AZ, USA), C. Baral, G. Brewka, and J. S. Schlipf, Eds. LNCS, vol. 4483. Springer, 201–213.
- MENEZES, L., VITORINO, J., AND AURELIO, M. 2005. A high performance CHR[∨] execution engine. In Schrijvers and Frühwirth (2005b), 35–45.

- MEYER, B. 2000. A constraint-based framework for diagrammatic reasoning. In Holzbaaur and Frühwirth (2000b), 327–344.
- MIRANKER, D. P. 1987. TREAT: a new and efficient match algorithm for AI production systems. Ph.D. thesis, Columbia Univ.
- MIRANKER, D. P. 1998. TREAT or RETE, neither, LEAPS is best. <http://www.cs.utexas.edu/~miranker/treator.htm>. Online note.
- MIRANKER, D. P. ET AL. 1991. Texas benchmark suite. <http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/>. First described in (Brant, Grose, Lofaso, and Miranker 1991).
- MIRANKER, D. P. AND BRANT, D. A. 1990. An algorithmic basis for integrating production systems and large databases. In *Proc. Sixth Intl. Conf. Data Engineering*. IEEE, 353–360.
- MIRANKER, D. P., BRANT, D. A., LOFASO, B. J., AND GADBOIS, D. 1990. On the performance of lazy matching in production systems. In *Proc. 8th Nat. Conf. Artif. Intell. (AAAI'90)*. AAAI Press, 685–692.
- MIRANKER, D. P. AND LOFASO, B. 1991. The organization and performance of a TREAT-based production system compiler. *IEEE Trans. Knowl. Data Eng.* 3, 1, 3–10.
- MORAWIETZ, F. 2000. Chart parsing and constraint programming. In *COLING '00: Proc. 18th Intl. Conf. on Computational Linguistics* (Saarbrücken, Germany), M. Kay, Ed. Morgan Kaufmann.
- MORAWIETZ, F. AND BLACHE, P. 2002. Parsing natural languages with CHR. Unpublished Draft.
- MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: engineering an efficient sat solver. In *DAC '01: Proc. 38th Design Automation Conf.* 530–535.
- MUGGLETON, S. AND RAEDT, L. D. 1994. Inductive Logic Programming: Theory and methods. *J. of Logic Programming* 19/20.
- OBERMEYER, L. AND MIRANKER, D. P. 1994. CLIPS++: Embedding CLIPS into C++. In *Proc. Third CLIPS Conference*.
- OBERMEYER, L., MIRANKER, D. P., AND BRANT, D. 1995. Selective indexing speeds production systems. In *Proc. 7th Intl. Conf. Tools with AI*. IEEE Comp. Society.
- PACHET, F. 1995. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming* 8, 4, 19–24.
- PACHET, F., Ed. 2004. *EOOPS'94: Proc. OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems*. Portland, Oregon, USA.
- PATON, N. W. AND DÍAZ, O. 1999. Active database systems. *ACM Computing Surveys* 31, 1, 63–103.
- PELOV, N. 2004. Semantics of logic programs with aggregates. Ph.D. thesis, K.U.Leuven, Belgium.
- PENN, G. 2000. Applying Constraint Handling Rules to HPSG. In Frühwirth et al. (2000).

- PILOZZI, P. 2009a. Automating termination proofs for CHR. In Hill and Warren (2009), 504–508.
- PILOZZI, P. 2009b. Proving termination by invariance relations. In Hill and Warren (2009), 499–503.
- PILOZZI, P. 2009c. Research summary: Termination of CHR. In Hill and Warren (2009), 534–535.
- PILOZZI, P. AND DE SCHREYE, D. 2008. Termination analysis of CHR revisited. In García de la Banda and Pontelli (2008), 501–515.
- PILOZZI, P. AND DE SCHREYE, D. 2009. Scaling termination proofs by a characterization of cycles in CHR. Tech. Rep. CW 541, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium. Apr.
- PRETSCHNER, A., SLOTSCH, O., AIGLSTORFER, E., AND KRIEBEL, S. 2004. Model-based testing for real. *J. Software Tools for Technology Transfer (STTT)* 5, 2–3, 140–157.
- PROBST, M. 2001. Proper tail recursion in C. Diplomarbeit, Institute of Computer Languages, Vienna University of Technology.
- RAISER, F. AND SNEYERS, J., Eds. 2009. *CHR '09: Proc. 6th Workshop on Constraint Handling Rules* (Pasadena, California). K.U.Leuven, Dept. Comp. Sc., Technical report CW 555.
- RAISER, F. AND TACCHELLA, P. 2007. On confluence of non-terminating CHR programs. In Djelloul, Duck, and Sulzmann (2007), 63–76.
- RIBEIRO, C., ZÚQUETE, A., FERREIRA, P., AND GUEDES, P. 2000. Security policy consistency. In Frühwirth et al. (2000).
- RILEY, G. ET AL. 2008. *CLIPS Reference Manual – Version 6.30 Beta*.
- RILEY, G. ET AL. 2010. CLIPS: A tool for building expert systems. <http://clipsrules.sourceforge.net/>.
- ROBIN, J. AND VITORINO, J. 2006. ORCAS: Towards a CHR-based model-driven framework of reusable reasoning components. In Fink, Tompits, and Woltran (2006), 192–199.
- ROBIN, J., VITORINO, J., AND WOLF, A. 2007. Constraint programming architectures: Review and a new proposal. *J. Universal Comp. Science* 13, 6, 701–720.
- ROSSI, F., VAN BEEK, P., AND WALSH, T., Eds. 2006. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence, vol. 1. Elsevier.
- SARNA-STAROSTA, B. AND RAMAKRISHNAN, C. 2007. Compiling Constraint Handling Rules for efficient tabled evaluation. In *PADL '07: Proc. 9th Intl. Symp. Practical Aspects of Declarative Languages* (Nice, France), M. Hanus, Ed. LNCS, vol. 4354. Springer, 170–184.
- SARNA-STAROSTA, B. AND SCHRIJVERS, T. 2008a. An efficient term representation for CHR indexing. In *CICLOPS '08: Proc. 8th Colloquium on Implementation of Constraint and Logic Programming Systems*, M. Carro and B. Demoen, Eds. 172–186.

- SARNA-STAROSTA, B. AND SCHRIJVERS, T. 2008b. Transformation-based indexing techniques for Constraint Handling Rules. In Schrijvers, Raiser, and Frühwirth (2008), 3–18.
- SCHIFFEL, S. AND THIELSCHER, M. 2007. Fluxplayer: A successful general game player. In *AAAI '07: Proc. 22nd AAAI Conf. Artificial Intelligence* (Vancouver, Canada). AAAI Press, 1191–1196.
- SCHINZ, M. AND ODERSKY, M. 2001. Tail call elimination on the Java Virtual Machine. In *BABEL'01: First Intl. Workshop on Multi-Language Infrastructure and Interoperability. ENTCS 59(1)*, 158–171.
- SCHMAUSS, M. 1999. An implementation of CHR in Java. Diplomarbeit, Institute of Computer Science, LMU, Munich, Germany.
- SCHRIJVERS, T. 2005. Analyses, optimizations and extensions of Constraint Handling Rules. Ph.D. thesis, K.U.Leuven, Belgium.
- SCHRIJVERS, T. 2008. The K.U.Leuven CHR System. <http://people.cs.kuleuven.be/~tom.schrijvers/CHR/>.
- SCHRIJVERS, T. AND BRUYNNOOGHE, M. 2006. Polymorphic algebraic data type reconstruction. In *PPDP '06*, A. Bossi and M. Maher, Eds. ACM Press, 85–96.
- SCHRIJVERS, T. AND DEMOEN, B. 2004a. Antimonotony-based delay avoidance for CHR. Tech. Rep. CW 385, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium. July.
- SCHRIJVERS, T. AND DEMOEN, B. 2004b. The K.U.Leuven CHR system: Implementation and application. In Frühwirth and Meister (2004), 8–12.
- SCHRIJVERS, T., DEMOEN, B., DUCK, G. J., STUCKEY, P. J., AND FRÜHWIRTH, T. 2006. Automatic implication checking for CHR constraints. In *RULE '05: 6th Intl. Workshop on Rule-Based Programming. ENTCS*, vol. 147(1). Elsevier, 93–111.
- SCHRIJVERS, T. AND FRÜHWIRTH, T. 2005a. Analysing the CHR implementation of union-find. In Wolf, Frühwirth, and Meister (2005), 135–146.
- SCHRIJVERS, T. AND FRÜHWIRTH, T., Eds. 2005b. *CHR '05: Proc. 2nd Workshop on Constraint Handling Rules* (Sitges, Spain). K.U.Leuven, Dept. Comp. Sc., Technical report CW 421.
- SCHRIJVERS, T. AND FRÜHWIRTH, T., Eds. 2006. *CHR '06: Proc. 3rd Workshop on Constraint Handling Rules* (Venice, Italy). K.U.Leuven, Dept. Comp. Sc., Technical report CW 452.
- SCHRIJVERS, T. AND FRÜHWIRTH, T. 2006. Optimal union-find in Constraint Handling Rules. *TPLP 6*, 1–2, 213–224.
- SCHRIJVERS, T. AND FRÜHWIRTH, T., Eds. 2008. *Constraint Handling Rules — Current Research Topics*. LNAI, vol. 5388. Springer.
- SCHRIJVERS, T., RAISER, F., AND FRÜHWIRTH, T., Eds. 2008. *CHR '08: Proc. 5th Workshop on Constraint Handling Rules* (Hagenberg, Austria). RISC Report Series 08-10, University of Linz, Austria.
- SCHRIJVERS, T., STUCKEY, P. J., AND DUCK, G. J. 2005. Abstract interpretation for Constraint Handling Rules. In *PPDP '05* (Lisbon, Portugal), P. Barahona and A. Felty, Eds. ACM Press, 218–229.

- SCHRIJVERS, T. AND WARREN, D. S. 2004. Constraint Handling Rules and tabled execution. In Demoen and Lifschitz (2004), 120–136.
- SCHRIJVERS, T., WARREN, D. S., AND DEMOEN, B. 2003. CHR for XSB. In *CICLOPS '03: Proc. 3rd Intl. Colloq. on Implementation of Constraint and Logic Programming Systems* (Mumbai, India), R. Lopes and M. Ferreira, Eds. University of Porto, Portugal, Dept. Comp. Sc., Technical report DCC-2003-05. 7–20.
- SCHRIJVERS, T., WIELEMAKER, J., AND DEMOEN, B. 2005. Poster: Constraint Handling Rules for SWI-Prolog. In Wolf, Frühwirth, and Meister (2005).
- SCHRIJVERS, T., ZHOU, N.-F., AND DEMOEN, B. 2006. Translating Constraint Handling Rules into Action Rules. In Schrijvers and Frühwirth (2006), 141–155.
- SCHUMANN, E. T. 2002. A literate programming system for logic programs with constraints. In *WFLP '02* (Grado, Italy), M. Comini and M. Falaschi, Eds. University of Udine, Research Report UDMI/18/2002/RR.
- SEITZ, C., BAUER, B., AND BERGER, M. 2002. Planning and scheduling in multi agent systems using Constraint Handling Rules. In *IC-AI '02: Proc. Intl. Conf. Artificial Intelligence* (Las Vegas, Nevada, USA). CSREA Press.
- SELMAN, D. ET AL. 2004. JSR-94 – JavaTM rule engine API. <http://jcp.org/en/jsr/summary?id=94>.
- SIMONS, P., NIEMELÁ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138, 1–2, 181–234.
- SNEYERS, J. 2008. Optimizing compilation and computational complexity of Constraint Handling Rules. Ph.D. thesis, K.U.Leuven, Belgium.
- SNEYERS, J., MEERT, W., AND VENNEKENS, J. 2009. CHRiSM: Chance Rules induce Statistical Models. In Raiser and Sneyers (2009), 62–76.
- SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006a. Dijkstra’s algorithm with Fibonacci heaps: An executable description in CHR. In Fink, Tompits, and Woltran (2006), 182–191.
- SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006b. Memory reuse for CHR. In Etalle and Truszczynski (2006), 72–86.
- SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2008. Guard reasoning in the refined operational semantics of CHR. In Schrijvers and Frühwirth (2008), 213–244.
- SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2009. The computational power and complexity of Constraint Handling Rules. *ACM TOPLAS* 31, 2 (Feb.).
- SNEYERS, J., VAN WEERT, P., AND SCHRIJVERS, T. 2007. Aggregates for Constraint Handling Rules. In Djelloul, Duck, and Sulzmann (2007), 91–105.
- SNEYERS, J., VAN WEERT, P., SCHRIJVERS, T., AND DE KONINCK, L. 2010. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. *TPLP* 10, 1, 1–47.
- SNEYERS, J., VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. 2007. Aggregates in CHR. Tech. Rep. CW 481, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium. Mar.
- STAHL, M. AND MELNIKOV, A. 2007. STMCHR. Available at the CHR Website (2010).

- STEINBRUNN, M., MOERKOTTE, G., AND KEMPER, A. 1997. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal* 6, 3, 191–208.
- STERLING, L. AND SHAPIRO, E. 1994. *The art of Prolog*, Second ed. MIT Press.
- STUCKEY, P. J. 1995. Negation and Constraint Logic Programming. *Information and Computation* 118, 1, 12–33.
- STUCKEY, P. J., Ed. 2002. *ICLP '02: Proc. 18th Intl. Conf. Logic Programming* (Copenhagen, Denmark). LNCS, vol. 2401. Springer.
- STUCKEY, P. J. AND SULZMANN, M. 2005. A theory of overloading. *ACM TOPLAS* 27, 6, 1216–1269.
- STUCKEY, P. J., SULZMANN, M., AND WAZNY, J. 2004. The Chameleon system. In Frühwirth and Meister (2004), 13–32.
- SULZMANN, M., DUCK, G. J., PEYTON-JONES, S., AND STUCKEY, P. J. 2007. Understanding functional dependencies via Constraint Handling Rules. *J. Functional Prog.* 17, 1, 83–129.
- SULZMANN, M. AND LAM, E. S. 2007a. Compiling Constraint Handling Rules with lazy and concurrent search techniques. In Djelloul, Duck, and Sulzmann (2007), 139–149.
- SULZMANN, M. AND LAM, E. S. 2007b. Haskell - Join - Rules. In *IFL '07: 19th Intl. Symp. Implementation and Application of Functional Languages* (Freiburg, Germany), O. Chitil, Ed. 195–210.
- SULZMANN, M. AND LAM, E. S. 2008. Parallel execution of multi-set constraint rewrite rules. In *PPDP '08* (Valencia, Spain), S. Antoy, Ed. ACM Press, 20–31.
- SULZMANN, M., LAM, E. S., AND VAN WEERT, P. 2008. Actors with multi-headed message receive patterns. In *COORDINATION '08: Proc. 10th Intl. Conf. Coordination Models and Languages* (Oslo, Norway), D. Lea and G. Zavattaro, Eds. Number 5052 in LNCS. Springer, 315–330.
- SULZMANN, M., SCHRIJVERS, T., AND STUCKEY, P. J. 2006. Principal type inference for GHC-style multi-parameter type classes. In *APLAS '06: Proc. 4th Asian Symp. on Programming Languages and Systems* (Sydney, Australia), N. Kobayashi, Ed. LNCS, vol. 4279. Springer, 26–43.
- SUN MICROSYSTEMS, INC. 2010. Java SE HotSpot at a glance. <http://java.sun.com/javase/technologies/hotspot/>.
- SWAMI, A. AND GUPTA, A. 1988. Optimization of large join queries. *SIGMOD Rec.* 17, 3, 8–17.
- TACCHELLA, P., GABBRIELLI, M., AND MEO, M. C. 2007. Unfolding in CHR. In Leuschel and Podelski (2007), 179–186.
- THIELSCHER, M. 2002. Reasoning about actions with CHRs and finite domain constraints. In Stuckey (2002), 70–84.
- THIELSCHER, M. 2005. FLUX: A logic programming method for reasoning agents. In Abdennadher, Frühwirth, and Holzbaur (2005), 533–565.
- VALDURIEZ, P. 1987. Join indices. *ACM Trans. Database Systems* 12, 2, 218–246.

- VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38, 3, 619–649.
- VAN HENTENRYCK, P. AND SARASWAT, V. 1997. Constraint Programming: strategic directions. *Constraints* 2, 1 (Apr.), 7–33.
- VAN WEERT, P. 2005. Constraint programming in Java: een gebruiksvriendelijk, flexibel en efficient CHR-systeem voor Java. M.S. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium. (In Dutch).
- VAN WEERT, P. 2006. *K.U.Leuven JCHR User's Manual*. Available at (Van Weert 2010c).
- VAN WEERT, P. 2008a. Compiling Constraint Handling Rules to Java: A reconstruction. Tech. Rep. CW 521, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium. Aug.
- VAN WEERT, P. 2008b. Optimization of CHR propagation rules. In García de la Banda and Pontelli (2008), 485–500.
- VAN WEERT, P. 2008c. A tale of histories. In Schrijvers, Raiser, and Frühwirth (2008), 79–94.
- VAN WEERT, P. 2010a. Efficient lazy evaluation of rule-based programs. *IEEE Transactions on Knowledge and Data Engineering*. Accepted.
- VAN WEERT, P. 2010b. Join ordering for constraint handling rules: Putting theory into practice. In *CHR'10: 7th International Workshop on Constraint Handling Rules*, P. Van Weert and L. De Koninck, Eds. To appear.
- VAN WEERT, P. 2010c. The K.U.Leuven JCHR system. <http://people.cs.kuleuven.be/~peter.vanweert/JCHR/>.
- VAN WEERT, P., DE KONINCK, L., AND SNEYERS, J. 2009. A proposal for a next generation of CHR. In Raiser and Sneyers (2009), 77–93.
- VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. 2005. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Schrijvers and Frühwirth (2005b), 47–62.
- VAN WEERT, P., SNEYERS, J., AND DEMOEN, B. 2008. Aggregates for CHR through program transformation. In King (2008), 59–73.
- VAN WEERT, P., SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006a. Extending CHR with negation as absence. In Schrijvers and Frühwirth (2006), 125–140.
- VAN WEERT, P., SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006b. To $\text{CHR}^{\bar{\square}}$ or not to $\text{CHR}^{\bar{\square}}$: Extending CHR with negation as absence. Tech. Rep. CW 446, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium. May.
- VAN WEERT, P., WUILLE, P., SCHRIJVERS, T., AND DEMOEN, B. 2008. CHR for imperative host languages. In Schrijvers and Frühwirth (2008), 161–212.
- VITORINO, J. AND AURELIO, M. 2005. CHORD. <http://chord.sourceforge.net/>.
- VOETS, D., PILOZZI, P., AND DE SCHREYE, D. 2007. A new approach to termination analysis of Constraint Handling Rules. In Djelloul, Duck, and Sulzmann (2007), 77–89.
- VOLL, K. 2006. A methodology of error detection: Improving speech recognition in radiology. Ph.D. thesis, Simon Fraser University, Burnaby, Canada.

- WALTZ, D. 1975. Understanding line drawing of scenes with shadows. In *The Psychology of Computer Vision*. McGraw-Hill, Chapter 2, 19–91.
- WANG, Y.-W. AND HANSON, E. N. 1992. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proc. 8th Intl. Conf. Data Engineering*. IEEE Comp. Society, 88–97.
- WIDOM, J. AND CERI, S. 1996. *Active database systems: triggers and rules for advanced database processing*. Morgan Kaufmann.
- WIELEMAKER, J. ET AL. 2010. SWI Prolog. <http://www.swi-prolog.org/>.
- WOLF, A. 1999. Adaptive Constraintverarbeitung mit Constraint-Handling-Rules – ein allgemeiner Ansatz zur Lösung dynamischer Constraint-probleme. Ph.D. thesis, Technical University Berlin, Germany.
- WOLF, A. 2000a. Projection in adaptive constraint handling. In Apt, Kakas, Monfroy, and Rossi (2000), 318–338.
- WOLF, A. 2000b. Toward a rule-based solution of dynamic constraint hierarchies over finite domains. In Frühwirth et al. (2000).
- WOLF, A. 2001a. Adaptive constraint handling with CHR in Java. In *CP '01* (Paphos, Cyprus), T. Walsh, Ed. LNCS, vol. 2239. Springer, 256–270.
- WOLF, A. 2001b. Attributed variables for dynamic constraint solving. In *Proc. 14th Intl. Conf. Applications of Prolog* (Tokyo, Japan). Prolog Assoc. of Japan, 211–219.
- WOLF, A. 2005. Intelligent search strategies based on adaptive Constraint Handling Rules. In Abdennadher, Frühwirth, and Holzbaur (2005), 567–594.
- WOLF, A., FRÜHWIRTH, T., AND MEISTER, M., Eds. 2005. *W(C)LP '05: Proc. 19th Workshop on (Constraint) Logic Programming* (Universität Ulm, Germany). Ulmer Informatik-Berichte, vol. 2005-01.
- WOLF, A., GRUENHAGEN, T., AND GESKE, U. 2000. On incremental adaptation of CHR derivations. In Holzbaur and Frühwirth (2000b), 389–416.
- WOLF, A., ROBIN, J., AND VITORINO, J. 2007. Adaptive CHR meets CHR[∇]: An extended refined operational semantics for CHR[∇] based on justifications. In Djelloul, Duck, and Sulzmann (2007), 1–15.
- WRIGHT, I. AND MARSHALL, J. 2003. The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *Intl. J. Intell. Games & Simul.* 2, 1, 36–48.
- WUILLE, P. 2007. CCHR: de snelste CHR implementatie. M.S. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium. In Dutch.
- WUILLE, P., SCHRIJVERS, T., AND DEMOEN, B. 2007. CCHR: the fastest CHR implementation, in C. In Djelloul, Duck, and Sulzmann (2007), 123–137.

Biography

Peter Van Weert was born on the 17th of January 1983 in Leuven, Belgium, and grew up in another Belgian city, Mechelen. After graduating from high school at the Scheppersinstituut Mechelen, his university studies took him back to his birth town. He studied computer science at the Faculty of Sciences of the K.U.Leuven university, receiving his Bachelor's degree of Science in Informatics (*Kandidaat Informatica*) in 2003, and his Master's degree of Science in Informatics (*Licentiaat Informatica*) in 2005. Both times, Peter graduated summa cum laude with the congratulations of the Board of Examiners. His Master's thesis, titled "Constraint Programming in Java: a user-friendly, flexible and efficient CHR system for Java", was supervised by prof. Bart Demoen.

In September 2005, Peter started working as Ph.D. student in the Security task force of the DistriNet research group at the department of Computer Science of the K.U.Leuven, under the supervision of Professor Frank Piessens. Shortly thereafter, he moved to the Analysis (*Design, Analysis and Implementation of Declarative Programming Languages*) subgroup of the DTAI (*Declarative Languages and Artificial Intelligence*) research group in January 2006. In October 2006, Peter became a research assistant funded by the Research Foundation – Flanders (FWO Vlaanderen). He was a visiting scholar at the National University of Singapore in November/December 2007 under supervision of prof. Martin Sulzmann.

List of Publications

Articles in International Reviewed Journals¹

VAN WEERT, P.. Efficient lazy evaluation of rule-based programs. *IEEE Transactions on Knowledge and Data Engineering* (impact factor: 2.236). To appear in *Special issue on Rule Representation, Interchange and Reasoning in Distributed, Heterogeneous Environments*, 2010.

SNEYERS, J., VAN WEERT, P., SCHRIJVERS, T., AND DE KONINCK, L.. As time goes by: Constraint Handling Rules — A survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming* 10, 1, 1–47, January 2010 (impact factor: 1.049).

VAN WEERT, P., WUILLE, P., SCHRIJVERS, T., AND DEMOEN, B.. CHR for imperative host languages. *Lecture Notes on Artificial Intelligence*, vol. 5388, pp. 161–212. Springer, December 2008.

Contributions at International Conferences and Symposia

VAN WEERT, P. Optimization of CHR propagation rules. In *ICLP '08*, M. García de la Banda and E. Pontelli, Eds. LNCS, vol. 5366, pp. 485–500 (full paper acceptance rate: 31.6%). Springer, 2008.

SULZMANN, M., LAM, E. S., AND VAN WEERT, P. Actors with multi-headed message receive patterns. In *COORDINATION '08: Proc. 10th Intl. Conf. Coordination Models and Languages*, D. Lea and G. Zavattaro, Eds. LNCS, vol. 5052, pp. 315–330 (acceptance rate: 34.4%). Springer, December 2008.

VAN WEERT, P., SNEYERS, J., AND DEMOEN, B. Aggregates for CHR through program transformation. In *LOPSTR '07, Revised Selected Papers*, A. King, Ed. LNCS, vol. 4915, pp. 59–73 (acceptance rate: 43.3%). Springer, 2008.

¹Journal impact factors are based on the 2008 edition of the Journal Citation Report[®] published by the Institute for Scientific Information (ISI).

SNEYERS, J., VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. Aggregates in Constraint Handling Rules: Extended abstract. In *ICLP '07*, V. Dahl and I. Niemelä, Eds. LNCS, vol. 4670. pp. 446–448. Springer, September 2007.

VAN WEERT, P.. Extension and implementation of CHR — Research summary. In *ICLP '07*, V. Dahl and I. Niemelä, Eds. LNCS, vol. 4670. pp. 466–468. Springer, September 2007.

Contributions at International Workshops

VAN WEERT, P., DE KONINCK, L., AND SNEYERS, J. A proposal for a next generation of CHR. In *CHR '09*, F. Raiser and J. Sneyers, Eds. K.U.Leuven, Dept. Comp. Sc., Technical report CW 555, pp. 77–93, July 2009.

VAN WEERT, P. A tale of histories. In *CHR '08*, T. Schrijvers, F. Raiser, and T. Frühwirth, Eds. RISC Report Series 08-10, University of Linz, Austria, Hagenberg, Austria, pp. 79–94, 2008.

SNEYERS, J., VAN WEERT, P., AND SCHRIJVERS, T. Aggregates for Constraint Handling Rules. In *CHR '07*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. Porto, Portugal, pp. 91–105, September 2007.

VAN WEERT, P., SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. Extending CHR with negation as absence. In *CHR '06*, T. Schrijvers and T. Frühwirth, Eds. K.U.Leuven, Dept. Comp. Sc., Technical report CW 452, Venice, Italy, 125–140, July 2006.

VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *CHR '05*, T. Schrijvers and T. Frühwirth, Eds. K.U.Leuven, Dept. Comp. Sc., Technical report CW 421, Sitges, Spain, 47–62, October 2005.

Technical Reports

VAN WEERT, P. Compiling Constraint Handling Rules to Java: A reconstruction. CW 521, K.U.Leuven, Dept. Computer Science. August 2008.

VAN WEERT, P. Optimization of CHR propagation rules: Extended report. CW 519, K.U.Leuven, Dept. Computer Science. August 2008.

SNEYERS, J., VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. Aggregates in CHR. CW 481, K.U.Leuven, Dept. Computer Science. March 2007.

VAN WEERT, P., SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. To CHR^{\neg} or not to CHR^{\neg} : Extending CHR with negation as absence. CW 446, K.U.Leuven, Dept. Computer Science. May 2006.

Arenberg Doctoral School of Science, Engineering & Technology

Faculty of Engineering

Department of Computer Science

DTAI (Declarative Languages and Artificial Intelligence)

Celestijnenlaan 200A box 2402

B-3001 Leuven

KATHOLIEKE UNIVERSITEIT
LEUVEN

