KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

Execution Control for Constraint Handling Rules

Promotor :
Prof. Dr. B. DEMOEN

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Leslie DE KONINCK

November 2008

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

Execution Control for Constraint Handling Rules

Jury :
Prof. Dr. ir. H. Van Brussel, voorzitter
Prof. Dr. B. Demoen, promotor
Prof. Dr. ir. M. Bruynooghe
Prof. Dr. ir. R. Cools
Prof. Dr. ir. G. Janssens
Prof. Dr. P. Stuckey (University of Melbourne)
Prof. Dr. M. Sulzmann (IT University of Copenhagen)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Leslie DE KONINCK

U.D.C. 681.3*D3

November 2008

D/2008/7515/103
ISBN 978–90–5682–993–3

Execution Control for Constraint Handling Rules

Abstract

Constraint Programming (CP) is a high-level declarative programming paradigm in which problems are modeled by means of constraints on the problem variables that need to hold in all solutions to the problem. Many problems of high practical relevance can easily be described in terms of constraints. Example application areas include production planning and crew scheduling. A constraint programming system contains a constraint solver whose task it is to find valuations for the problem variables that satisfy all constraints.

Constraint programming systems can be classified by the variable domains and types of constraints their solver supports. However, for many problems it is not so straightforward to create a model in terms of basic constraint domains. Therefore, on the one hand, systems emerged that can handle more specialized constraint domains. On the other hand, facilities were designed that make it easier to implement an application-specific constraint solver. Some notable examples of these facilities are attributed variables, and Constraint Handling Rules.

Constraint Handling Rules (CHR) is a rule-based language, designed for the implementation of application-specific constraint solvers. CHR is a very flexible language for the specification of a constraint solvers' logic, but flexible execution control is almost completely lacking. Execution control is of great importance for the efficiency of CP systems. However, so far, the problem of execution control has received only very limited attention in the context of CHR. In this thesis, we propose a solution to the problem of execution control in CHR. In particular, we extend CHR with high-level facilities to support the specification of execution strategies.

More precisely, we extend CHR with user-defined rule priorities into CHR^{rp}. CHR rules correspond to constraint propagators, and so rule priorities enable the specification of propagation strategies. An optimized implementation of CHR^{rp} is presented and evaluated empirically. Next, we extend CHR with facilities for search strategy control. Our approach combines $\text{CHR}^{\vee}$ (CHR with disjunction) and CHR^{rp} into a new language called $\text{CHR}^{\text{brp}}_{\vee}$ in which the propagation strategy is determined by means of rule priorities, and the search strategy by means of branch priorities. We propose a framework for analyzing the time complexity of CHR^{rp} programs, by combining the Logical Algorithms framework (LA) of Ganzinger and McAllester with CHR^{rp}. We present translation schemas from and to LA and propose an alternative implementation for CHR^{rp} with strong complexity guarantees. Finally, we investigate the join order optimization problem, which is

an important aspect of optimized CHR compilation. We propose a cost model for matching multi-headed rules, approximations of its parameters, and methods to find an optimal join order. An extension of the model for CHR^{rp} is given.

Uitvoeringscontrole voor Constraint Handling Rules

Beknopte Samenvatting

Constraint Programming (CP) is een hoog-niveau declaratief programmeerparadigma waarbij problemen gemodelleerd worden door middel van constraints (beperkingen) op de probleemvariabelen waaraan alle oplossingen moeten voldoen. Vele problemen van hoge praktische relevantie kunnen eenvoudig beschreven worden via constraints. Zo zijn er toepassingen te vinden in onder andere productieplanning en het opstellen van dienstroosters voor personeel. Een constraint programming systeem omvat een constraint solver. Deze heeft als taak waarderingen voor de probleemvariabelen te vinden die aan alle constraints voldoen.

Constraint Programming systemen worden traditioneel opgedeeld op basis van het domein van de variabelen en het type van de ondersteunde constraints. Vele praktische problemen zijn echter moeilijk te modelleren in termen van de standaard constraintdomeinen. Daarom zijn er langs de ene kant systemen ontwikkeld voor meer gespecialiseerde domeinen. Langs de andere kant zijn er faciliteiten ontwikkeld die het eenvoudiger maken om gespecialiseerde constraint solvers te implementeren. Voorbeelden daarvan zijn geattribueerde variabelen en Constraint Handling Rules.

Constraint Handling Rules (CHR) is een regelgebaseerde taal, ontworpen voor de implementatie van gespecialiseerde constraint solvers. CHR is een zeer flexibele taal voor de specificatie van de logica van een constraint solver. Flexibele uitvoeringscontrole ontbreekt echter bijna volledig. Uitvoeringscontrole is van fundamenteel belang voor de efficiëntie van CP systemen. Tot dusver is hier echter binnen de context van CHR nauwelijks aandacht aan besteed. In dit proefschrift presenteren we een oplossing voor het probleem van uitvoeringscontrole in CHR. In het bijzonder breiden we CHR uit met hoog-niveau faciliteiten om de specificatie van uitvoeringsstrategieën te ondersteunen.

Meer specifiek breiden we CHR uit met gebruikersgedefinieerde regelprioriteiten, tot CHR^{rp}. CHR regels komen overeen met constraint propagatoren, en regelprioriteiten maken dus de specificatie van propagatiestrategieën mogelijk. Een geoptimaliseerde implementatie van CHR^{rp} wordt voorgesteld en empirisch geëvalueerd. Vervolgens breiden we CHR uit met faciliteiten voor de controle van de zoekstrategie. Onze aanpak combineert $\text{CHR}^{\vee}$ (CHR met disjunctie) en CHR^{rp} tot een nieuwe taal, $\text{CHR}^{\text{brp}}_{\vee}$ genaamd, waarin de propagatiestrategie bepaald is d.m.v. regelprioriteiten, en de zoekstrategie d.m.v. vertakkingsprioriteiten. We stellen een raamwerk voor de analyse van de tijdscomplexiteit van CHR^{rp} programma's voor, door het Logical Algorithms raamwerk (LA) van Ganzinger en

McAllester te combineren met CHR^{rp}. We presenteren vertalingsschema's van en naar LA en stellen een alternatieve implementatie voor CHR^{rp} voor, die sterke complexiteitsgaranties biedt. Tenslotte onderzoeken we het join order optimalisatieprobleem, hetgeen een belangrijk deel is van geoptimaliseerde CHR compilatie. We presenteren een kostenmodel voor het matchen met meerhoofdige regels, evenals benaderingen voor zijn parameters en methodes om een optimale join order te vinden. Een uitbreiding van het kostenmodel voor CHR^{rp} wordt gegeven.

Acknowledgements

First and foremost, I thank my supervisor, Bart Demoen, for taking me on as his PhD student, for guiding me to the finish, for advertising my work in Melbourne, and so much more. I thank the other members of the jury for reading and commenting on my text: chairman Hendrik van Brussel; the remaining members of my guidance committee: Maurice Bruynooghe and Ronald Cools; Gerda Janssens, Peter Stuckey and Martin Sulzmann.

In particular, I thank Maurice for taking the time to try and understand every bit of text, which, given that he does the same for all PhD students of DTAI is an extraordinary achievement. I thank Ronald for taking time to read a text that must be far outside of his areas of interest. Gerda has been my main connection to educational tasks during this PhD: I assisted with declarative languages exercise sessions a few semesters, and I was the daily advisor of two Masters thesis students of which she was the supervisor. The latter has allowed me to get experience on some of the more practical aspects of CHR. It has certainly been an interesting and sometimes fun experience, for which I express my gratitude.

I thank Peter for inviting me to Melbourne, twice. Those were amongst the best months of my life and I can hardly wait to go back to Down Under. I thank him for showing interest in my work, both during my stays at Melbourne University, and as a member of my PhD jury. I am also grateful to him for encouraging me to work on ACD Term Rewriting which is a fascinating topic. I have only met Martin on a few occasions; I remember his enthusiasm when presenting research results and think he is a great motivator.

During the course of this PhD, I had the pleasure to work together with some great co-authors. Apart from Bart and Peter (Stuckey), these are Tom Schrijvers, Gregory Duck, Jon Sneyers and Peter Van Weert. Tom was my Masters thesis advisor and continued to take on this advisor role, especially during the first years of my PhD. Tom also started the CHR research here in Leuven, and so basically, without him, this thesis would not exist. I am grateful to Tom for making all of this possible. Gregory showed me around in Melbourne (well he mostly showed me where to eat near the ICT building) and taught me about the peculiarities of ACD Term Rewriting and about what matters for the efficient compilation of

it. Apart from that, we talked about many other things during lunch breaks at strange places like a garage where they sell Indonesian food.

Jon and Peter are my current office mates. Before that, Jon and I were joined by Tom in the office on the other side of the corridor, but we moved so that Tom could more closely follow up on 'the new guy' Pieter Wuille. Jon first introduced me to the DTAI Alma group. He stopped going to the Alma some time ago, which, given that he is a vegetarian, is completely understandable: you can only eat vegetarian lasagna so many times. These days I am the only representative of the analysis subgroup during lunch, but I enjoy the company of all those ML and KRR people anyway ;-). Now we're at it, I'd like to thank Daan Fierens for all his help on the practical side of finishing a PhD.

Jon and I have a bit of a coffee addiction. When our caffeine levels have sunk below a certain threshold, one of us will raise our coffee cup and say "Coffee?", followed by a trip to the cafeteria. Peter does not drink coffee, but often joins us anyway. Apart from keeping us awake, the coffee breaks mostly have a social function. Jon, who started his PhD a year earlier than me, was to me often a standard to evaluate my own progress by. I eventually did finish my PhD a bit earlier than him, but that's probably because I'm too impatient and because I haven't been renovating a house in the meanwhile. Peter joined our CHR group about half a year later than me. We were fellow students during our pre-graduate studies, but did not have much contact back then. Peter likes to tinker with his work until it's perfect. I am fortunately or unfortunately (depending on one's perspective) not such a perfectionist.

I thank the administration and system administration staff for making this department run smoothly. From August 2005 until December 2005, I worked on the project "TREND^3: Towards reliable and enhanced network design and deployment for DSL", funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), and in collaboration with Alcatel. Since January 2006, my research has been funded by a PhD grant of IWT-Vlaanderen.

In Melbourne, I stayed at Melbourne University's Graduate House. My first visit overlapped with that of Quan Phan. I thank him, and also his wife Lan for the fun time we had together. I also greatly enjoyed the friendship from my fellow Grad House residents: thanks Shang-Hui, Yuin Inn, Julien, Petra, Makenzie, Ilya, Joonha, Colin, Kai and so many others.[1]

Finally, I'd like to thank my family and friends for their support and for everything else not related to this thesis.

[1] This list may be biased towards my second visit, which was longer and obviously more recent.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Execution Control for CHR

Constraint Programming (CP) (Apt 2003) is a high-level declarative programming paradigm in which problems are modeled by means of constraints, that is, relations on the problem variables that need to hold in all solutions to the problem. Many problems of high practical relevance can easily be described in terms of constraints. Example application areas include production planning, crew scheduling, design and control of electrical circuits and natural language processing. A constraint programming system contains a constraint solver whose task is to find valuations for the problem variables that satisfy all constraints.

Constraint programming systems can be classified by the variable domains and types of constraints their solver supports, for example constraints over finite domains, or linear constraints over the real numbers. However, for many problems it is not so straightforward to create a model in terms of such basic constraint domains. Therefore, on the one hand, systems emerged that can handle more specialized constraint domains, such as constraints over finite sets (Dovier et al. 2000) and constraints over graphs (Dooms et al. 2005). On the other hand, facilities were designed that make it easier to implement an application-specific constraint solver. In the context of Constraint Logic Programming (CLP) (Jaffar and Maher 1994), a combination of constraint programming and logic programming, some notable developments are indexicals (Carlsson et al. 1997), attributed variables (Holzbaur 1992), and Constraint Handling Rules (Frühwirth 1998).

Constraint Handling Rules (CHR) is a rule-based language, specifically designed for the implementation of application-specific constraint solvers. Language concepts such as multi-headed rules, guards and multi-set semantics, make CHR a very flexible language for the specification of a constraint solvers' logic. However, flexible execution control is almost completely lacking. Execution control

is of great importance for the efficiency of C(L)P systems, see e.g. (Schulte and Stuckey 2004; Ringwelski and Hoche 2005). However, so far, the problem of execution control has received only very limited attention in the context of CHR. In this thesis, we propose a solution to the problem of execution control in CHR. In particular, we extend CHR with high-level facilities to support the specification of execution strategies. It is our aim to offer the flexibility needed to support the type of execution control needed for efficient constraint solving, while limiting both the number of new language constructs the CHR programmer has to learn, as well as the (runtime) overhead introduced by a more flexible execution mechanism.

1.2 Goal and Contributions

The main goal of this thesis is the design, implementation, and analysis of language concepts and constructs that allow for a flexible, efficient and high-level form of execution control in Constraint Handling Rules. Constraint solving generally consists of both *propagation* and *search*. During propagation, information from different constraints is combined, which may lead to simplification of constraints and the removal of inconsistent values from the domains of the problem variables. During search, the problem variables are labeled, that is, they are speculatively assigned values from their domains. The separation of constraint solving into propagation and search also leads to a corresponding separation of the execution strategy into a propagation strategy and a search strategy.

Now we give a summary of our main contributions. A more complete overview can be found at the end of each of the main chapters, and in the general conclusion in Chapter 7. Our first two contributions consist in the design and implementation of language extensions for CHR to support the specification of propagation and search strategies:

- In Chapter 3, we extend CHR with user-defined rule priorities into CHR^{rp}. CHR rules correspond to constraint propagators: they are used to simplify combinations of constraints, as well as to make implicit information available explicitly in the form of a (logically redundant) constraint. Therefore, rule priorities enable the specification of propagation strategies. We show how CHR^{rp} can be used to concisely express strategies that lead to more efficient programs. An optimized implementation of CHR^{rp} is presented and evaluated empirically.

- Chapter 4 extends the work of the previous chapter with facilities for search strategy control. Regular CHR does not offer support for search, but a language extension called $\text{CHR}^{\vee}$ does. Our approach combines $\text{CHR}^{\vee}$ and CHR^{rp} into a new language called $\text{CHR}^{\text{brp}}_{\vee}$ in which the propagation strategy is determined by means of rule priorities, and the search strategy by

means of branch priorities. We show how various common search strategies can be specified in $\text{CHR}^{\text{brp}}_{\vee}$ and how such strategies can be combined. We also present an extension of the language to support conflict-directed backjumping.

Our remaining contributions are as follows:

- The CHR^{rp} language extension was inspired by a related bottom-up logic programming language presented by Ganzinger and McAllester (2002). In that paper, the *Logical Algorithms*[1] language is the basis for a meta-complexity result for logic programming languages that gives the time complexity of a logic program in terms of the number of rule firings and other concepts at the language level (rather than requiring an analysis of low-level implementation details). In Chapter 5, we formally establish the correspondence between CHR^{rp} and Logical Algorithms, and present an extension of the Logical Algorithms meta-complexity result to support a high-level way of analyzing the time complexity of CHR^{rp} programs.

- The task of join order optimization is an important part of the optimized compilation of CHR programs. It consists in finding an optimal order to match the heads of a multi-headed CHR rule with the constraints in the constraint store. In Chapter 6, we present the first formal and in-depth study of the join ordering problem. More precisely, we propose a model for the cost of matching multi-headed rules, present estimates for its parameters, and show how to minimize the cost according to this model. This work applies to CHR in general; a refinement of the cost model is given for the specific case of CHR^{rp}.

Chapter 2 contains preliminaries on CHR and Chapter 7 presents a general conclusion for this thesis.

1.3 Bibliographical Note

Most parts of this thesis have been published before:

- Chapter 2 is mostly based on preliminaries of the publications that are the basis for the later chapters. Furthermore, Section 2.3 is taken from

  De Koninck, L. 2007. Mergeable schedules for lazy matching. Tech. Rep. CW 505, Department of Computer Science, K.U.Leuven.

  and Section 2.7 comes from

[1] The language was not given a name by Ganzinger and McAllester (2002), but the paper itself is called "Logical Algorithms".

Sneyers, J., Van Weert, P., Schrijvers, T., and De Koninck, L. 2008. As time goes by: Constraint Handling Rules: A survey of CHR research between 1998 and 2007. Submitted to Theory and Practice of Logic Programming.

- Chapter 3 is based on

  De Koninck, L., Stuckey, P. J., and Duck, G. J. 2008. Optimizing compilation of CHR with rule priorities. In *9th International Symposium on Functional and Logic Programming*, J. Garrigue and M. Hermenegildo, Eds. Lecture Notes in Computer Science, vol. 4989. Springer, 32–47.

  De Koninck, L., Schrijvers, T., and Demoen, B. 2007. User-definable rule priorities for CHR. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, M. Leuschel and A. Podelski, Eds. ACM Press, 25–36.

  De Koninck, L., Schrijvers, T., and Demoen, B. 2007. CHR$^{\text{rp}}$: Constraint Handling Rules with rule priorities. Tech. Rep. CW 479, Department of Computer Science, K.U.Leuven.

- Chapter 4 corresponds to

  De Koninck, L., Schrijvers, T., and Demoen, B. 2008. A flexible search framework for CHR. To appear in Lecture Notes in Artificial Intelligence: Special Issue on Recent Advances in Constraint Handling Rules.

- Chapter 5 is based on

  De Koninck, L., Schrijvers, T., and Demoen, B. 2007. The correspondence between the Logical Algorithms language and CHR. In *23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 209–223.

- Finally, Chapter 6 is based on

  De Koninck, L. and Sneyers, J. 2007. Join ordering for Constraint Handling Rules. In *4th Workshop on Constraint Handling Rules*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. U.Porto, 107–121.

During the course of this PhD, I have also worked on two other topics. The first topic builds further on my Masters thesis in which a constraint solver capable of solving nonlinear polynomial constraints over the real numbers was developed. The system is based on interval computations and is implemented using CHR. This work resulted in the publication

De Koninck, L., Schrijvers, T., and Demoen, B. 2006. INCLP(ℝ) - Interval-based nonlinear constraint logic programming over the reals. In *20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. INFSYS Research Report 1843-06-02. TU Wien, 91–100.

I also worked on the implementation of ACD Term Rewriting, a term rewriting language that extends both CHR and AC term rewriting (Duck et al. 2006):

Duck, G. J., De Koninck, L., and Stuckey, P. J. 2008. Cadmium: Implementing ACD term rewriting. Accepted at the 24th International Conference on Logic Programming.

Chapter 2

Background: Constraint Handling Rules

This chapter introduces background material on the Constraint Handling Rules (CHR) language. We assume the reader is somewhat familiar with logic programming and Prolog, but we do not assume any familiarity with CHR. In Section 2.1, we provide some notational conventions. Section 2.2 introduces the syntax and semantics of CHR. CHR rules can be multi-headed and in Section 2.3 it is shown how rule instances for such multi-headed rules can be found. In Section 2.4, we present the refined operational semantics of CHR, which captures the basic execution mechanism used by most current implementations of CHR. Section 2.5 gives some further preliminaries on the compilation of CHR into Prolog. We discuss the program properties of confluence and termination in Section 2.6, and present a selection of applications of CHR in Section 2.7.

2.1 Notation

This section presents some notation used throughout the text. In code examples, we use the syntax of Prolog and CHR with Prolog as its host language. This means that variable names start with an uppercase letter, and functors with a lowercase letter. Furthermore, all verbatim code is in typewriter font, whereas italic font is used for variables at the source-code level (i.e., not for program variables). The latter is mainly used to represent program transformations and generic code. Throughout the text, we make use of some Prolog built-ins. Table 2.1 lists the most important ones.

A sequence of elements $e_1, \ldots, e_n$ is denoted by $[e_1, \ldots, e_n]$. The empty sequence is denoted by ϵ and sequence concatenation by $+\!\!+$. We also use the Prolog-style notation $[H \mid T]$ to denote $[H] +\!\!+ T$.

Predicates on lists

- `member`(X, L): X is an element of list L
- `select`(X, L_1, L_2): X is an element of L_1 and L_2 is a list of L_1's other elements
- `append`(L_1, L_2, L_3): list L_3 results from appending list L_2 to list L_1
- `length`(L, N): list L contains N elements

Higher order predicates

- `call`(G): G (meta-call; converts a term to an atom)
- `findall`(T, G, L): L contains an element $\theta(T)$ for each answer $\theta(G)$ of goal G

Arithmetic predicates and functions

- X `is` Y: arithmetic expression Y evaluates to X
- X `=:=` Y: arithmetic expressions X and Y evaluate to the same value
- X `=\=` Y: arithmetic expressions X and Y evaluate to a different value
- `abs`(X): the absolute value of X
- X `mod` Y: X modulo Y
- `min`(X, Y): the minimum of X and Y

Input and output

- `write`(T): writes term T on the output stream
- `read`(T): reads term T from the input stream
- `nl`: writes a newline character on the output stream

Analyzing terms

- `nonvar`(T): term T is not a variable
- `ground`(T): term T is ground (i.e., T does not contain variables)

Table 2.1: Prolog built-ins used throughout the text

2.2 Constraint Handling Rules

Constraint Handling Rules (CHR) (Frühwirth 1998) is a high-level rule-based programming language, designed for the purpose of facilitating the implementation of application-tailored constraint solvers. CHRs are multi-headed and guarded rewrite rules that operate on a multi-set of user-defined constraints called the CHR constraint store. A CHR constraint solver, also called a constraint handler, runs on top of a host language which offers it a built-in constraint solver. The first CHR implementations had Prolog as the host language (Holzbaur and Frühwirth 1999; Holzbaur and Frühwirth 2000; Schrijvers and Demoen 2004). Later, implementations emerged for, amongst others, the functional programming language Haskell (Chin et al. 2003; Lam and Sulzmann 2007), the multi-paradigm languages Curry (Hanus 2006) and HAL (Holzbaur et al. 2005), as well as the imperative languages Java (Abdennadher et al. 2002; Van Weert et al. 2005) and C (Wuille et al. 2007) (see also (Van Weert et al. 2008)).

CHR rules implement constraint simplification and propagation. They allow the definition of user-defined constraints in terms of lower-level built-in constraints which are subsequently solved by constraint solvers offered by the host language. For example, CHR implementations with Prolog as the host language may use the Prolog unification algorithm to solve equality constraints over Herbrand terms.

Example 2.1 (Less-or-Equal) The less-or-equal (`leq`) program is a classic CHR example. It implements a less-than-or-equal constraint by eventually translating it into equality constraints. It illustrates the three basic rule types of CHR, namely *simplification*, *simpagation* and *propagation*. The rules of `leq` are shown in Listing 2.1.

```
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 2.1: `leq` program in CHR

The first rule has as name `reflexivity` (before `@`) and is a *simplification* rule that states that any constraint of the form `leq`(a,a) should be 'simplified' to (i.e. replaced by) `true`. The second rule, `antisymmetry`, states that two constraints `leq`(a,b) and `leq`(b,a) should be replaced with $a = b$, constraining the arguments to be equal. The third rule (`idempotence`) is a *simpagation* rule that says that given two constraints of the form `leq`(a,b) we should replace the second one (after the \) by `true`. Simpagation rules replace the constraints after the backslash by the constraints in the body, given the constraints before the backslash. They can be considered as a combination of constraint simplification and propagation (hence their name).

In CHR, constraints have multi-set semantics, which means that the multiplicity of constraints is important. The `idempotence` rule basically states that `leq`/2 constraints have set semantics. Finally, the fourth rule (`transitivity`) is a *propagation* rule, which says given constraints `leq`(a,b) and `leq`(b,c) we should add a new constraint `leq`(a,c) without deleting anything. □

In the next subsections, we formally define the syntax and semantics of CHR. We consider both the declarative semantics and the operational semantics.

2.2.1 Syntax and Declarative Semantics

A constraint $c(t_1, \ldots, t_n)$ is an atom of predicate c/n with t_i a host language value (e.g., a Herbrand term in Prolog) for $1 \leq i \leq n$. There are two types of constraints: built-in constraints and CHR constraints (also called user-defined constraints). The CHR constraints are solved by the CHR program whereas the built-in constraints are solved by an underlying constraint solver (e.g., the Prolog unification algorithm).

There are three types of Constraint Handling Rules: *simplification rules*, *propagation rules* and *simpagation rules*. They have the following form:

$$\begin{array}{lllll}
\textbf{Simplification} & r\ @ & H^r & \iff & g \mid B \\
\textbf{Propagation} & r\ @\ H^k & & \implies & g \mid B \\
\textbf{Simpagation} & r\ @\ H^k \setminus H^r & & \iff & g \mid B
\end{array}$$

where r is the rule *name*, H^k and H^r are sequences of CHR constraints and are called the *heads* of the rule, the rule *guard* g is a conjunction of built-in constraints, and the rule *body* B is a multi-set of both CHR and built-in constraints. The sequences H^k and H^r are non-empty except that H^k is empty in simplification rules and H^r is empty in propagation rules. Throughout the text, in particular in the descriptions of the operational semantics, we use the simpagation rule form to denote any type of rule. A program P is a set of CHR rules.

Let $\mathsf{vars}(A)$ be the variables occurring in A, let $\bar{\exists}_A F$ denote $\exists x_1, \ldots, \exists x_n F$ where $\{x_1, \ldots, x_n\} = \mathsf{vars}(F) \setminus \mathsf{vars}(A)$ and let $\forall_A F$ denote $\forall y_1, \ldots, \forall y_n F$ where $\{y_1, \ldots, y_n\} = \mathsf{vars}(A)$. Let $H = \langle H^k, H^r, g \rangle$. A CHR program P forms a constraint theory consisting of the constraint theory of the built-in constraint solver in conjunction with one of the following formulas for each rule r in P (depending on its type):

$$\begin{array}{ll}
\textbf{Simplification} & \forall_H((\quad g) \rightarrow (H^r \leftrightarrow \bar{\exists}_H B)) \\
\textbf{Propagation} & \forall_H((H^k \wedge g) \rightarrow (\quad \bar{\exists}_H B)) \\
\textbf{Simpagation} & \forall_H((H^k \wedge g) \rightarrow (H^r \leftrightarrow \bar{\exists}_H B))
\end{array}$$

1. Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_t}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_t}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. Apply $\langle G, H_1 \uplus H_2 \uplus S, B, T\rangle_n \overset{\omega_t}{\mapsto}_P \langle C \uplus G, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where P contains a (renamed apart) rule $$r \ @ \ H_1' \backslash H_2' \iff g \mid C$$ and a matching substitution θ such that $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and $t = \langle \mathsf{id}(H_1), \mathsf{id}(H_2), r\rangle \notin T$.

Table 2.2: Transitions of ω_t

2.2.2 Operational Semantics

Operationally, CHR constraints have multi-set semantics. To distinguish between different occurrences of syntactically equal constraints, CHR constraints are extended with a unique identifier. An identified CHR constraint is denoted by $c\#i$ with c a CHR constraint and i the identifier. We write $\mathsf{chr}(c\#i) = c$ and $\mathsf{id}(c\#i) = i$. An alternative declarative semantics for CHR based on intuitionistic linear logic, that amongst others takes into account this multi-set semantics, is given by Betz and Frühwirth (2005).

The *theoretical operational semantics*[1] of CHR, denoted ω_t, is given by Duck et al. (2004) as a state transition system. A CHR execution state σ is represented as a tuple $\langle G, S, B, T\rangle_n$ where G is the goal, a multi-set of constraints that need to be solved; S is the CHR constraint store, a set of identified CHR constraints; B is the built-in constraint store, a conjunction of built-in constraints; T is the propagation history, a set of tuples denoting the rule instances that have already fired; and n is the next free identifier, used to identify new CHR constraints. The transitions of ω_t are shown in Table 2.2, where $\mathcal{D}$ denotes the built-in constraint theory and $\uplus$ is the multi-set union operation which we also use for sets in case a disjoint union is required.

The **Solve** transition solves a built-in constraint from the goal, the **Introduce** transition inserts a new CHR constraint from the goal into the CHR constraint store, and the **Apply** transition fires a rule instance. A rule instance $\theta(r)$ instantiates a rule with CHR constraints matching the heads, using matching substitution θ. The propagation history prevents trivial non-termination by ensuring that each rule instance (in particular if the rule is a propagation rule) fires at most once.

A CHR derivation starts from an initial state of the form $\langle G, \emptyset, \mathtt{true}, \emptyset\rangle_1$ with G the goal: a multi-set of constraints to be solved. We implicitly assume that no ω_t transition applies to a state in which the built-in constraint store is inconsistent,

[1] Other operational semantics exist that reduce the non-determinism of the theoretical operational semantics; see Section 2.4 and Section 3.3.2.

i.e., states of the form $\langle G, S, B, T\rangle_n$ with $\mathcal{D} \models \neg\bar{\exists}_{\emptyset} B$.[2] A state σ is called *final* if no more transitions apply to it. If its built-in constraint store is inconsistent, it is called a *failed* final execution state; otherwise, σ is a *successful* final execution state. We write $\sigma \overset{\omega}{\not\rightarrowtail}_P$ if σ is a final state for program P and under operational semantics ω. Similarly, we write $\sigma_1 \overset{\omega}{\rightarrowtail}{}^*_P \sigma_n$ to denote the existence of a derivation from state σ_1 to state σ_n, consisting of $n-1$ transitions $\sigma_1 \overset{\omega}{\rightarrowtail}_P \sigma_2 \overset{\omega}{\rightarrowtail}_P \ldots \overset{\omega}{\rightarrowtail}_P \sigma_n$.

Example 2.2 An example derivation for the `leq` program given in Listing 2.1 and initial goal $G = \{\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(B,A)}\}$ is shown below:

$$
\begin{aligned}
& \langle\{\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(B,A)}\}, \emptyset, \texttt{true}, \emptyset\rangle_1 \\
\textbf{Introduce} \overset{\omega_t}{\rightarrowtail}_{\texttt{leq}} & \langle\{\texttt{leq(B,C)}, \texttt{leq(B,A)}\}, \{\texttt{leq(A,B)\#1}\}, \texttt{true}, \emptyset\rangle_2 \\
\textbf{Introduce} \overset{\omega_t}{\rightarrowtail}_{\texttt{leq}} & \langle\{\texttt{leq(B,A)}\}, \{\texttt{leq(A,B)\#1}, \texttt{leq(B,C)\#2}\}, \texttt{true}, \emptyset\rangle_3 \\
\textbf{Introduce} \overset{\omega_t}{\rightarrowtail}_{\texttt{leq}} & \langle\emptyset, \{\texttt{leq(A,B)\#1}, \texttt{leq(B,C)\#2}, \texttt{leq(B,A)\#3}\}, \texttt{true}, \emptyset\rangle_4 \\
\textbf{Apply} \overset{\omega_t}{\rightarrowtail}_{\texttt{leq}} & \langle\{\texttt{A} = \texttt{B}\}, \{\texttt{leq(B,C)\#2}\}, \texttt{true}, \emptyset\rangle_4 \\
\textbf{Solve} \overset{\omega_t}{\rightarrowtail}_{\texttt{leq}} & \langle\emptyset, \{\texttt{leq(B,C)\#2}\}, \texttt{A} = \texttt{B}, \emptyset\rangle_4
\end{aligned}
$$

□

We define the (multi-)set of *qualified answers* for a goal G, given program P and operational semantics ω, as follows:

$$\mathcal{QA}(G) = \left\{ S \uplus B \mid \langle G, \emptyset, \texttt{true}, \emptyset\rangle_1 \overset{\omega}{\rightarrowtail}{}^*_P \langle G', S, B, T\rangle_n \overset{\omega}{\not\rightarrowtail}_P \right\}$$

Here, G' must be the empty set if B is consistent. Each element in $\mathcal{QA}(G)$ is referred to as a qualified answer of G. A qualified answer can be considered as a multi-set or conjunction of constraints, depending on the context.

2.3 Matching Algorithms

The task of matching consists of finding constraints that instantiate the rule heads in such a way that the (implicit and explicit) guard is entailed by the built-in store. We call such an instantiation of the rule heads a rule instance and denote it by $\theta(r)$ with r the rule and θ a matching substitution. A *partial match* instantiates part of the rule heads; a rule instance corresponds to a *full match*. In this subsection, we take a closer look to how the matching task is performed.

[2] We make the same assumption in the description of other operational semantics of CHR and its extensions.

2.3.1 Pattern Matching in Rule-Based Languages

Pattern matching for multi-headed rules (like the ones in CHR) has traditionally been studied in the context of production rule systems. Notable developments in this area are the RETE algorithm by Forgy (1982), and the TREAT (Miranker 1987) and LEAPS (Miranker et al. 1990) algorithms. The former two algorithms implement eager (*data driven*) matching, while the latter implements lazy (*demand driven*) matching. The difference is that in eager matching, all rule instances are generated first and then one of them is fired, whereas in lazy matching, rule instances are generated incrementally and on demand, one at a time.

The RETE algorithm keeps track of partial rule matches in so-called *beta memories*. Whenever a new data element (e.g., a constraint in the CHR context) is inserted, it is used to extend already found partial matches into either larger partial matches or full matches. If rules cannot have negated heads (which is the case in CHR), then an element deletion is handled by removing those partial and full matches in which the deleted element participates. After all added and removed elements have been processed, a full match is selected and the corresponding rule instance is fired.

The TREAT algorithm works in a similar fashion, except that it does not store partial matches. Instead, full matches are generated by iteratively adding heads starting from a newly inserted data element, which is called the *dominant object* in this role. Deletion then only requires removing those *full* matches in which the deleted element participates. Finally, LEAPS is like TREAT, but stops generating matches as soon as a full match is found, after which the corresponding rule instance is fired immediately. Afterwards, the matching process continues if the dominant object has not been deleted by then. In LEAPS, no matches are ever stored. The main advantage of LEAPS, and to a lesser extent also of TREAT, are the lower memory requirements by not having to store partial and/or full matches. However, the laziness of LEAPS in itself has the advantage that no computation is wasted on producing matches that are invalidated before being used. This suggests that also a lazy version of RETE (i.e., lazy matching *with* storage of partial matches that are computed on the way) makes sense (see e.g. (Proctor 2006) and Section 5.5) although in general it would consume considerably more memory than LEAPS matching.

2.3.2 Matching in Constraint Handling Rules

In CHR, rules need to be matched with a multi-set of CHR constraints, which more or less correspond to the data elements in production rule systems. An interesting aspect of CHR in this context is that it supports non-ground CHR constraints, i.e., constraints whose arguments contain variables. These variables can be further constrained (or instantiated) by means of the built-in constraints which may enable new rule firings. The effect of built-in constraints could be

implemented by first removing the affected CHR constraints, and then reasserting the updated versions of them.

Most current CHR implementations use a variant of the LEAPS algorithm to perform the rule matching task. Instead of actually removing and reasserting CHR constraints that are affected by a built-in constraint, these constraints are 'reactivated' after the update, i.e., they become dominant objects again. As said before, the operational semantics of CHR requires that each rule fires at most once for each combination of CHR constraints. This requirement is enforced by maintaining a *propagation history* consisting of tuples representing the rule instances that already fired. Note that the propagation history does not avoid any redundant matching work. Also, in the worst case, the history takes space proportional to the maximal number of applicable rule instances at any time.[3] However, in (Van Weert 2008) it is shown that maintaining a propagation history is often not necessary.

2.4 The Refined Operational Semantics

The refined operational semantics of CHR, denoted by ω_r, is introduced in (Duck et al. 2004) as a formalization of the execution mechanism of most current CHR implementations. While its initial purpose was descriptive, it has become a normative description of how CHR implementations should work. This is because many CHR programs are written with the ω_r semantics in mind and often they do not work correctly under the ω_t semantics (i.e., they are not confluent). In Chapter 3, we present a more high-level alternative to the ω_r semantics that still allows considerable execution control.

The ω_r semantics is based on lazy matching (cf. Section 2.3) and as such on the concept of an *active* constraint (dominant object). The active constraint is a CHR constraint that is used as a starting point for finding rule instances. To ensure that all rule instances are eventually tried, all CHR constraints introduced in the goal or a rule body are scheduled (on a stack) to be activated. CHR constraints that have been active before and whose variables are affected by a new built-in constraint, are scheduled to be reactivated.

The active constraint tries rules in textual order until either it finds an applicable rule instance or all rules have been tried. When a rule instance fires, the constraints in its body are processed from left to right. This may cause the active constraint to be temporarily suspended while other constraints are activated. After processing the rule body, the active constraint resumes its search for a next applicable rule instance under the condition that it has not been removed meanwhile. If the active constraint *has* been removed, or no more applicable rule

[3] If the propagation history is not cleared of tuples involving removed constraints, it may even become larger.

instances can be found, processing resumes where it left before the current active constraint was activated.

Execution states of the ω_r semantics are of the form $\langle A, S, B, T\rangle_n$ where the CHR constraint store S, built-in constraint store B, propagation history T, and next free identifier n are as in the ω_t semantics. The *execution stack* A contains the constraints to be solved, similar to the goal G under ω_t, as well as active CHR constraints, which are of the form $c\#i : j$ and represent an identified constraint $c\#i$ trying to find rule instances in which it matches the j^{th} occurrence of the predicate of c in the program. Occurrence numbers are assigned from top to bottom and within each rule from right to left.[4] For example, Listing 2.2 shows the `leq` program of Listing 2.1 with each head annotated with an occurrence number (in subscript). An *active* constraint $\texttt{leq}(x, y)\#i : 3$ with i some identifier, will try to find rule instances in which it matches the left-most head of the `antisymmetry` rule. Only the topmost active constraint in A actively looks for rule instances. The other ones are temporarily suspended.

```
reflexivity  @ leq(X,X)1 <=> true.
antisymmetry @ leq(X,Y)3, leq(Y,X)2 <=> X = Y.
idempotence  @ leq(X,Y)5 \ leq(X,Y)4 <=> true.
transitivity @ leq(X,Y)7, leq(Y,Z)6 ==> leq(X,Z).
```

Listing 2.2: `leq` program in CHR with occurrence numbers shown

Now, the ω_r semantics is again given as a state transition system, whose transitions are shown in Table 2.3. The **Solve** and **Activate** transitions are similar to the **Solve** and **Introduce** transitions in the ω_t semantics. The main difference is that both transitions add CHR constraints on the execution stack for the purpose of activating them. The reactivation of CHR constraints after solving a built-in constraint that affects the variables of the constraints in question, is done in two steps: the **Solve** transition adds the *identified* constraints to the execution stack, and the **Reactivate** transition converts the topmost identified CHR constraint into an *active* constraint. Note that the **Solve** transition as defined here reactivates at least all constraints whose variables are not fixed by the built-in constraint store (essentially, all non-ground constraints). In (Schrijvers 2005), a lower is given on which constraints need to be reactivated.

The **Simplify** and **Propagate** transitions correspond to the ω_t **Apply** transition, but only considers rules in which the active constraint matches a specific head, namely the j^{th} occurrence of the predicate of c if $c\#i : j$ is the active constraint (the *active occurrence*). The **Default** transition tries the next occurrence of the predicate of c if no more rule instances can be found in which the active constraint matches the active occurrence. Finally, the **Drop** transition makes the

[4] This ensures a rule's removed occurrences are tried before its kept ones.

1. Solve $\langle [c \mid A], S_0 \cup S_1, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle S_1 +\!\!+ A, S_0 \cup S_1, c \wedge B, T\rangle_n$ where c is a built-in constraint and $\mathsf{vars}(S_0) \subseteq \mathsf{fixed}(B)$ where $\mathsf{fixed}(B)$ are the variables whose value is fixed by B.
2. Activate $\langle [c \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle [c\#n : 1 \mid A], \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. Reactivate $\langle [c\#i \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle [c\#i : 1 \mid A], S, B, T\rangle_n$ where c is a CHR constraint.
4. Drop $\langle [c\#i : j \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle A, S, B, T\rangle_n$ if CHR constraint c has no j^{th} occurrence in P.
5. Simplify $\langle [c\#i : j \mid A], H_1 \uplus H_2 \uplus \{c\#i\} \uplus H_3 \uplus S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle C +\!\!+ A, H_1 \cup S, \theta \wedge B, T\rangle_n$ where the j^{th} occurrence of c in a (renamed apart) rule in P is $$r \ @\ H_1' \backslash H_2', d_j, H_3' \iff g \mid C$$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathsf{chr}(H_3) = \theta(H_3')$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$.
6. Propagate $\langle [c\#i : j \mid A], H_1 \uplus \{c\#i\} \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle C +\!\!+ [c\#i : j \mid A], H_1 \cup \{c\#i\} \cup H^2 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where the j^{th} occurrence of c in a rule in P is $$r \ @\ H_1', d_j, H_2' \backslash H_3' \iff g \mid C$$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathsf{chr}(H_3) = \theta(H_3')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and $t = \langle \mathsf{id}(H_1) +\!\!+ [i] +\!\!+ \mathsf{id}(H_2), \mathsf{id}(H_3), r\rangle \notin T$.
7. Default $\langle [c\#i : j \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle [c\#i : j+1 \mid A], S, B, T\rangle_n$ if no other transition applies.

Table 2.3: Transitions of ω_r

active constraint passive if all occurrences of its constraint symbol have been tried.

Example 2.3 An example derivation under the ω_r semantics for the same program and goal as in Example 2.2, is shown in Figure 2.1. □

2.5 Implementation Preliminaries

In this section, we give some preliminaries concerning the representation and storage of CHR constraints in Prolog-based CHR implementations. The purpose of this section is to make the presentation of the CHR$^{\text{rp}}$ implementations in Section 3.6 and Section 5.5 more comprehensible.

CHR constraints are represented as Prolog terms. In the most basic representation, an identified CHR constraint $c\#i$ could be represented as a term

$$
\begin{aligned}
& \langle [\mathtt{leq(A,B)}, \mathtt{leq(B,C)}, \mathtt{leq(B,A)}], \emptyset, \mathtt{true}, \emptyset \rangle_1 \\
\textbf{Activate} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(A,B)}\#1 : 1, \mathtt{leq(B,C)}, \mathtt{leq(B,A)}], \{\mathtt{leq(A,B)}\#1\}, \mathtt{true}, \emptyset \rangle_2 \\
\textbf{Default}_{\times 7} \; \textbf{Drop} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,C)}, \mathtt{leq(B,A)}], \{\mathtt{leq(A,B)}\#1\}, \mathtt{true}, \emptyset \rangle_2 \\
\textbf{Activate} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,C)}\#2 : 1, \mathtt{leq(B,A)}], \\
& \qquad \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2\}, \mathtt{true}, \emptyset \rangle_3 \\
\textbf{Default}_{\times 5} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,C)}\#2 : 6, \mathtt{leq(B,A)}], \\
& \qquad \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2\}, \mathtt{true}, \emptyset \rangle_3 \\
\textbf{Propagate} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(A,C)}, \mathtt{leq(B,C)}\#2 : 6, \mathtt{leq(B,A)}], \\
& \qquad \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2\}, \mathtt{true}, T \rangle_3 \\
\textbf{Activate} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(A,C)}\#3 : 1, \mathtt{leq(B,C)}\#2 : 6, \mathtt{leq(B,A)}], \\
& \qquad \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3\}, \mathtt{true}, T \rangle_4 \\
\textbf{Default}_{\times 7} \; \textbf{Drop} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,C)}\#2 : 6, \mathtt{leq(B,A)}], \\
& \qquad \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3\}, \mathtt{true}, T \rangle_4 \\
\textbf{Default}_{\times 2} \; \textbf{Drop} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,A)}], \{\mathtt{leq(A,B)}\#1, \\
& \qquad \mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3\}, \mathtt{true}, T \rangle_4 \\
\textbf{Activate} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,A)}\#4 : 1], \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2, \\
& \qquad \mathtt{leq(A,C)}\#3, \mathtt{leq(B,A)}\#4\}, \mathtt{true}, T \rangle_5 \\
\textbf{Default} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,A)}\#4 : 2], \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2, \\
& \qquad \mathtt{leq(A,C)}\#3, \mathtt{leq(B,A)}\#4\}, \mathtt{true}, T \rangle_5 \\
\textbf{Simplify} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{A} = \mathtt{B}], \{\mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3\}, \mathtt{true}, T \rangle_5 \\
\textbf{Solve} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3], \\
& \qquad \{\mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3\}, \mathtt{A} = \mathtt{B}, T \rangle_5 \\
\textbf{Reactivate} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,C)}\#2 : 1, \mathtt{leq(A,C)}\#3], \\
& \qquad \{\mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3\}, \mathtt{A} = \mathtt{B}, T \rangle_5 \\
\textbf{Default}_{\times 3} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(B,C)}\#2 : 4, \mathtt{leq(A,C)}\#3], \\
& \qquad \{\mathtt{leq(B,C)}\#2, \mathtt{leq(A,C)}\#3\}, \mathtt{A} = \mathtt{B}, T \rangle_5 \\
\textbf{Simplify} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{true}, \mathtt{leq(A,C)}\#3], \{\mathtt{leq(A,C)}\#3\}, \mathtt{A} = \mathtt{B}, T \rangle_5 \\
\textbf{Solve} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(A,C)}\#3], \{\mathtt{leq(A,C)}\#3\}, \mathtt{A} = \mathtt{B}, T \rangle_5 \\
\textbf{Reactivate} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle [\mathtt{leq(A,C)}\#3 : 1], \{\mathtt{leq(A,C)}\#3\}, \mathtt{A} = \mathtt{B}, T \rangle_5 \\
\textbf{Default}_{\times 7} \; \textbf{Drop} \quad & \overset{\omega_r}{\mapsto}_{\mathtt{leq}} \langle \epsilon, \{\mathtt{leq(A,C)}\#3\}, \mathtt{A} = \mathtt{B}, T \rangle_5
\end{aligned}
$$

where $T = \{\langle [1,2], \epsilon, \mathtt{transitivity} \rangle\}$

Figure 2.1: Example derivation under the ω_r semantics

`suspension`(c, i).[5] In practical implementations, other fields (arguments) are added to the representation, including a state, parts of the propagation history involving the constraint (*distributed propagation history*) and pointers for index management. The term representation of a CHR constraint is often called its *suspension term*.

The suspension terms of the CHR constraints are stored in various indexes to allow for lookups during matching, as well as to support reactivating the CHR constraints affected by a built-in constraint (ω_r **Solve** transition). In the CHR implementation for Prolog by Holzbaur and Frühwirth (1999), as well as later systems such as the K.U.Leuven CHR system (Schrijvers and Demoen 2004) and our CHR$^{\text{rp}}$ implementation (Chapter 3), (part of) the constraint store is implemented by means of attributed variables (Holzbaur 1992; Demoen 2002). Attributed variables are Prolog variables that are associated with a term called its attribute. By means of built-in predicates, one can retrieve and (destructively) update the attribute of a given variable. Whenever an attributed variable is bound to another attributed variable, or to a term (during unification), a user-defined *unification hook* predicate is called, in which the affected variables' attributes can be processed.

Attributed variables can be used to implement (part of) the CHR constraint store as follows: all CHR constraints in which a given variable appears, have their suspension terms attached to this variable, i.e., the variable has an attribute which is a list of the suspension terms in which it appears. In a unification hook, the suspension term lists attached to the variables involved are merged and attached to the variables that remain after binding. Moreover, the constraints in these suspension term lists are reactivated as required by the ω_r **Solve** transition.

The representation of the CHR constraint store by means of attributed variables also allows for fast lookup of constraints during matching. Indeed, if we know a variable that appears in the constraint to be looked up, then this constraint must be attached to this variable. Building further on this idea, the CHR implementation described in (Holzbaur and Frühwirth 1999) attaches *all* CHR constraints to the attribute of a special variable that is globally accessible. This way, constraints can be looked up during matching, regardless of whether they share variables with the already computed partial match. Apart from the attributed variables-based constraint indexes, modern CHR implementations such as the K.U.Leuven CHR system also offer specialized indexes (e.g., hash tables or arrays (Sneyers et al. 2006a)) for retrieving constraints given a ground subset of their arguments.

Inserting constraints into the relevant indexes can be a relatively expensive operation, even though it normally only takes a constant amount of time per constraint (amortized and on average, for example when using hash tables for

[5] The functor name comes from the use of attributed variables to store constraints, where the constraints are considered as suspended goals that need to be executed when the variables in question are instantiated further. See (Holzbaur and Frühwirth 1999).

indexing). It is for this reason that modern CHR systems try to defer storage of a constraint as long as possible; this is called *late storage*, see e.g. (Holzbaur et al. 2005). Some constraints are always removed before reaching the point at which they need to be stored. For example, consider the following rules for retrieving the domain of a variable, represented by a `domain`/2 constraint.

```
domain(X,DX) \ get_domain(X,QDX) <=> QDX = DX.
get_domain(_,_) <=> fail.
```

Under the refined operational semantics of CHR, we have that once a `get_domain`/2 constraint becomes active, it remains active until it is removed, either by the first rule because a corresponding `domain`/2 constraint exists, or unconditionally by the second rule otherwise. Therefore, `get_domain`/2 constraints are *never stored*. Moreover, an active `domain`/2 constraint will never be able to find a matching `get_domain`/2 constraint, and so we can skip the occurrence of the `domain`/2 constraint in the first rule. We call such an occurrence that can be skipped, a *passive occurrence*. Schrijvers et al. (2005) propose an analysis to derive when constraints are to be stored, if ever, and which occurrences can be made passive. The analysis is based on an abstract interpretation framework for CHR, presented in the same paper, and tries to derive at which point other constraints can *observe* a given constraint. In Section 3.7, we propose a specialized form of late storage, called *late indexing*, as well as an analysis for passive occurrences, in the context of CHR$^{\text{rp}}$: CHR extended with rule priorities.

2.6 Program Properties

This section discusses some important properties of CHR programs, and how to derive them. More precisely, we look at *confluence* (Section 2.6.1) and *termination* (Section 2.6.2).

2.6.1 Confluence

Abdennadher (1997) gives a criterion for deciding whether a (terminating) CHR program is confluent under the ω_t semantics. Intuitively, confluence means that the answers to a goal are independent of the actual execution path followed. More formally, confluence is based on the concept of joinability:

Definition 1 (Joinability) *Two states σ_1 and σ_2 are joinable given program P and operational semantics ω if $\sigma_1 \overset{\omega}{\longmapsto}{}^*_P \sigma_1^*$ and $\sigma_2 \overset{\omega}{\longmapsto}{}^*_P \sigma_2^*$ with σ_1^* and σ_2^* variants with respect to the common variables of σ_1 and σ_2.*

Two execution states are variants with respect to a set of variables $\mathcal{V}$ if they are identical modulo renaming of constraint identifiers, and after projection of the

built-in constraints on the variables in $\mathcal{V}$ and removal of propagation history tuples involving removed CHR constraints.

Definition 2 (Local Confluence) *A CHR program P is locally confluent under operational semantics ω if for all states σ, σ_1 and σ_2: if $\sigma \overset{\omega}{\longmapsto}_P \sigma_1$ and $\sigma \overset{\omega}{\longmapsto}_P \sigma_2$ then σ_1 and σ_2 are joinable.*

If P terminates then local confluence implies confluence.[6]

Definition 3 (Confluence) *A CHR program P is confluent under operational semantics ω if for all states σ, σ_1 and σ_2: if $\sigma \overset{\omega}{\longmapsto}^*_P \sigma_1$ and $\sigma \overset{\omega}{\longmapsto}^*_P \sigma_2$ then σ_1 and σ_2 are joinable.*

Under ω_t, local confluence can be proven by checking all *critical pairs* for joinability. A critical pair consists of two *minimal* states σ_1 and σ_2 which follow from firing respectively rule instance $\theta_1(r_1)$ and $\theta_2(r_2)$ in common ancestor state σ such that $\theta_2(r_2)$ cannot fire in σ_1 and $\theta_1(r_1)$ cannot fire in σ_2.

Under the refined operational semantics, the critical pair test does not work because a derivation that is possible in a minimal state, may not be possible in a larger state due to the execution order imposed by the ω_r semantics. In (Duck 2005, Chapter 6), a practical test is developed to decide whether a program is confluent under ω_r. Unlike the critical pair test, this practical test is not complete. Note that every program that is confluent under ω_t, is also confluent under ω_r, whereas the opposite does not hold. Finally, a program that is non-confluent, may in practice still give the same answers, regardless of the derivation path, because the execution states leading to non-confluence cannot occur in practice. This issue is dealt with in (Duck et al. 2007), where the concept of *observable confluence* is introduced. A program is observable confluent if it is confluent for all execution states that are reachable from a valid initial state. The concept is generalized into $\mathcal{I}$-confluence, which is confluence w.r.t. a given invariant $\mathcal{I}$.

2.6.2 Termination

The first work on termination analysis of CHR programs is presented by Frühwirth (2000). In this work, it is demonstrated that termination proof techniques from logic programming and term rewrite systems can be adapted to the CHR context. Termination of CHR programs is proved by defining a ranking function from computation states to a well-founded domain such that the rank of consecutive computation states decreases. A condition on simplification rules guarantees such rank decreases for all consecutive states. This approach, however, cannot prove termination of CHR programs with propagation rules, because it is impossible to show decreases between consecutive states as these rules do not remove constraints from the store.

[6] Confluence for non-terminating programs is considered in (Raiser and Tacchella 2007).

Recently, three new results on termination analysis of CHR were presented. A termination preserving transformation of CHR programs to Prolog programs is described by Pilozzi et al. (2007). By reusing termination tools from logic programming and, indirectly, from term rewriting, proofs of termination of the transformed CHR programs are generated automatically, yielding the first fully automatic termination prover for CHR. The transformation, however, does not consider propagation histories. As such, it is applicable only to CHR programs without propagation rules. A transformation of single-headed propagation rules to equivalent simplification rules overcomes this problem partially.

The second contribution is presented in (Voets et al. 2007) and applies to a much larger class of CHR programs, compared to previous approaches. By formulating a new termination condition that verifies conditions imposed on the dynamic process of adding constraints to the store, the authors derive conditions for both simplification and propagation rules. However, for programs with simplification rules only, the approach of (Frühwirth 2000) gives stronger results.

Finally, Pilozzi and De Schreye (2008) present a termination condition that is more powerful than both the one in (Frühwirth 2000) and the one in (Voets et al. 2007). The condition is based on a new representation for CHR states that takes into account both the token store (dual of the propagation history, see Section 2.2.2) and the constraints that would result from further applying propagation rules until exhaustion.

2.7 Applications

In this section, we present some application areas of CHR. More precisely, we have made a selection of the applications presented in (Sneyers et al. 2008, Section 7). We discuss how CHR is used for the implementation of constraint solvers and classic algorithms, for type checking and type inference, abduction, and computational linguistics. Other application areas include multi-agent systems, the semantic web, and testing and verification (see (Sneyers et al. 2008, Section 7) for more details).

2.7.1 Constraint Solvers

CHR was originally designed specifically for writing constraint solvers. We discuss some recent examples of constraint solvers written in CHR. The following examples illustrate how CHR can be used to build effective prototypes of non-trivial constraint solvers:

Rational Trees Meister et al. (2006) present a solver for existentially quantified conjunctions of non-flat equations over rational trees. The solver consists of a transformation to flat equations, after which a classic CHR solver for

rational trees can be used. This results in a complexity improvement with respect to previous work. Djelloul et al. (2007) use the solver for rational trees as part of a more general solver for (quantified) first-order constraints over finite and infinite trees.

Non-linear Constraints A general purpose CHR-based CLP system for non-linear (polynomial) constraints over the real numbers is presented by De Koninck et al. (2006). The system, called INCLP($\mathbb{R}$), is based on interval arithmetic and uses an interval Newton method as well as constraint inversion to achieve respectively box and hull consistency.

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, performing acquisition of domain elements only when necessary.

Solvers Derived from Union-find Frühwirth (2006) proposes linear-time algorithms for solving certain Boolean equations and linear polynomial equations in two variables. These solvers are derived from the classic union-find algorithm (see Section 2.7.2).

Soft Constraints An important class of constraints are the so-called *soft constraints* which are used to represent preferences amongst solutions to a problem. Unlike hard (required) constraints which must hold in any solution, soft (preferential) constraints must only be satisfied as far as possible. Bistarelli et al. (2004) present a series of constraint solvers for (mostly) extensionally defined finite domain soft constraints, based on the framework of *c-semirings*. In this framework, soft constraints can be combined and projected onto a subset of their variables, by using the two operators of a c-semiring. A node and arc consistency solver is presented, as well as complete solvers based on variable elimination or branch and bound optimization.

Another well-known formalism for describing over-constrained systems is that of *constraint hierarchies*, where constraints with hierarchical strengths or preferences can be specified, and non-trivial error functions can be used to determine the solutions. Wolf (2000) proposes an approach for solving dynamically changing constraint hierarchies. Constraint hierarchies over finite domains are transformed into equivalent constraint systems, which are then solved using an adaptive CHR solver (see e.g. (Wolf et al. 2000)).

Scheduling Abdennadher and Marte (2000) have successfully used CHR for scheduling courses at the university of Munich. Their approach is based on a form of soft constraints, implemented in CHR, to deal with teachers' preferences. A related problem, namely that of assigning classrooms to courses given a

timetable, is dealt with in (Abdennadher et al. 2000). An overview of both applications is found in (Abdennadher 2001).

2.7.2 Union-Find and Other Classic Algorithms

CHR is used increasingly as a general-purpose programming language. Starting a trend of investigating this side of CHR, Schrijvers and Frühwirth (2006) implemented and analyzed 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. This work lead to parallel versions of the union-find algorithm (Frühwirth 2005) and several derived algorithms (Frühwirth 2006). Inspired by the specific optimal complexity result for the union-find algorithm, Sneyers et al. (2008) have generalized this to arbitrary (RAM-machine) algorithms.

The question of finding elegant and natural implementations of classic algorithms in CHR remains nevertheless an interesting research topic. Examples of recent work in this area are implementations of Dijkstra's shortest path algorithm using Fibonacci heaps (Sneyers et al. 2006a) and the preflow-push maximal flow algorithm (Meister 2006).

2.7.3 Programming Language Development

Another application area in which CHR has proved to be useful is the development of programming languages. CHR has been applied to implement additional programming language features such as type systems and abduction. Also, CHR has been used to implement new programming languages, especially in the context of computational linguistics.

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 no more than a simple Herbrand equality constraint, more advanced type systems require custom constraint solvers.

Alves and Florido (2002) presented the first work on using Prolog and CHR for implementing the type inference framework HM(X), i.e., type inference for extensions of the Hindley-Milner type system. This work was followed up by TypeTool (Hugo Simões and Florido 2004), a tool for visualizing the type inference process.

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. Moreover, various extensions, such as multi-parameter type classes (Sulzmann et al. 2006) and functional dependencies (Sulzmann et al. 2007) are easily expressed.

Coquery and Fages (2003, 2005) presented TCLP, a CHR-based type checker for Prolog and CHR on top of 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) present a 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, including diagnosis, recognition, natural language processing and type inference.

The earliest paper connecting CHR with abduction is by Abdennadher and Christiansen (2000). It shows how to model logic programs with abducibles and integrity constraints in CHR$^\vee$. The disjunction is used for (naively) enumerating the alternatives of the abducibles, while integrity constraints are implemented as propagation rules. The HYPROLOG system of Christiansen and Dahl (2005) combines the above approach to abductive reasoning with abductive-based logic programming in one system. Both the abducibles and the assumptions are implemented as CHR constraints. Christiansen (2006) also proposes the use of CHR for the implementation of *global abduction*, an extended form of logical abduction for reasoning about a dynamic world.

Gavanelli et al. (2003) propose two variant approaches to implementing abductive logic programming. The first is similar to the above approach, but tries to leverage as much as possible from a more efficient Boolean constraint solver, rather than CHR. The second approach propagates abducibles based on integrity constraints. The system of Alberti et al. (2005) extends the abductive reasoning procedure with the dynamic acquisition of new facts. These new facts serve to confirm or disconfirm earlier hypotheses. Finally, Sulzmann et al. (2005) show that advanced type system features give rise to implications of constraints, or, in other words, constraint abduction. An extension of their CHR-based type checking algorithm is required to deal with these implications.

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 earlier). It is therefore not surprising that CHR has proven a powerful implementation and specification tool for language processors.

Penn (2000) focuses on another benefit CHR provides to computational linguists, namely the possibility to delay constraints until their arguments are suf-

ficiently 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 for a flexible and clear implementation of a series of standard chart parsing algorithms (see also (Morawietz 2000)), as well as more advanced grammar formalisms such as minimalist grammars and property grammars. Items of a conventional chart parser are modeled as CHR constraints, and new constraints are deduced using constraint propagation. The constraint store represents the chart, from which the parse tree can be determined. Along the same lines is the CHR implementation of a context-sensitive, rule-based grammar formalism in (Garat and Wonsever 2002).

A more 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 in (Christiansen 2005). Contrary to the aforementioned approaches, which mostly use CHR as a general-purpose implementation language, Christiansen recognizes that the CHR language itself can be used as a powerful grammar formalism. CHRGs, built as a relatively transparent layer of syntactic sugar over CHR, are to CHR what DCGs are to Prolog.

CHRG's inherent support for context-sensitive rules readily allows linguistic phenomena such as long-distance reference and coordination to be modeled naturally (Christiansen 2005; Aguilar-Solis and Dahl 2004; Dahl 2004). CHRG grammar rules can also use extra-grammatical hypotheses, modeled as regular CHR constraints. This caters e.g. for straightforward implementations of assumption grammars and abductive language interpretation with integrity constraints.

2.8 Further Reading

For a more extensive overview of the Constraint Handling Rules language, its theoretical properties, implementation and applications, we refer to the 1998 survey (Frühwirth 1998), as well as the more recent survey (Sneyers et al. 2008), part of which was developed during the course of this PhD. We also refer to the PhD theses of Schrijvers (2005) and Duck (2005) which contain amongst others a thorough study of the implementation of CHR.

Chapter 3

Constraint Handling Rules with Rule Priorities

This chapter introduces CHR^{rp}: Constraint Handling Rules with rule priorities. CHR^{rp} offers flexible control of the propagation strategy which is lacking in CHR. A formal operational semantics for the extended language is given and is shown to be an instance of the theoretical operational semantics of CHR. We relate rule priorities to alternative forms of execution control. We show that CHR^{rp} can be efficiently compiled into the underlying host language. To achieve that, we introduce various optimizations, whose effects are empirically evaluated. The CHR^{rp} compiler is also compared with the state-of-the-art K.U.Leuven CHR system and shown to exhibit comparable performance while offering a much more high-level form of execution control.

3.1 Introduction

CHR is very flexible with respect to the specification of program logic, but it lacks high-level facilities for execution control. In particular, the control flow is most often fixed by the call-stack based refined operational semantics of CHR (see Section 2.2.2). To change the default execution strategy in implementations based on the refined semantics, one has to use auxiliary constructs like flag constraints, i.e., constraints whose sole purpose is to facilitate execution control. Such an approach is neither flexible nor efficient.

In this chapter, we propose extending CHR with user-definable rule priorities. The extension, called CHR^{rp}, offers the flexible execution control needed for implementing highly efficient Constraint (Logic) Programming systems. Moreover, it facilitates implementing rule-based algorithms as these often require certain rules to be tried before others, either for efficiency reasons, or for correctness. Rule

priorities allow the programmer to write programs that are more concise and efficient, but potentially not confluent under the theoretical operational semantics of CHR. They support a high-level form of execution control that is more declarative, flexible and comprehensible than previously available, while retaining the expressive power needed for the implementation of general purpose algorithms. In CHR$^{\text{rp}}$, all execution control information is explicit in the rule priority annotations, which leads to a clear separation of the logic and control aspects of a CHR$^{\text{rp}}$ program in line with (Kowalski 1979). We first give some examples of CHR with rule priorities.

Example 3.1 (Less-or-Equal) As a first example, we add priorities to the `leq` program of Example 2.1. The result is shown in Listing 3.1. Note that other priority assignments are also possible.

```
1 :: reflexivity  @ leq(X,X) <=> true.
1 :: antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
1 :: idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
2 :: transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Listing 3.1: `leq` program in CHR$^{\text{rp}}$

The rule priorities are given before the `::` symbol. The first three rules have a priority of 1, while the last rule has a priority of 2. By convention, lower numbers denote higher priorities, and so this priority assignment prefers constraint simplification (simpagation) over propagation. In fact, in the theoretical operational semantics of CHR, this program is not guaranteed to terminate because for some queries, e.g. $\{\texttt{leq}(\texttt{X},\texttt{X}), \texttt{leq}(\texttt{X},\texttt{Y})\}$, the `transitivity` rule may fire infinitely many times. The above shown priority assignment ensures the program terminates under the operational semantics of CHR$^{\text{rp}}$ because the `transitivity` rule is only allowed to fire when no other rule can. □

Dynamic rule priorities allow the priority of a rule to depend on the variables occurring on the left hand side of the rule.

Example 3.2 (Dijkstra's Shortest Path) A CHR$^{\text{rp}}$ implementation of Dijkstra's single-source shortest path algorithm is given in Listing 3.2.

```
  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).
```

Listing 3.2: Dijkstra's shortest path algorithm in CHR$^{\text{rp}}$

The input consists of a set of directed weighted edges, represented as `edge`/3 constraints where the first and last arguments respectively denote the begin and

end nodes, and the middle argument represents the weight. The source node is given by the `source`/1 constraint. The algorithm keeps track of upper-bounds on the shortest path distances to the different nodes, represented by `dist`/2 constraint whose arguments are respectively the node in question and the distance. Eventually, the distance upper-bounds become tight.

The first rule initiates the algorithm by creating a (tight) distance upper-bound to the source node of zero. The second rule removes redundant distance upper-bounds. Both these rules have a *static* priority of 1. Finally, the last rule has a *dynamic* priority and generates new distance upper-bounds. The priorities ensure these upper-bounds are created in the order required by Dijkstra's algorithm. □

The rest of this chapter is organized as follows. Section 3.2 presents further motivation for extending CHR with rule priorities. In Section 3.3, we formalize the syntax and semantics of the extended language: CHR$^{\text{rp}}$. Two important properties of CHR$^{\text{rp}}$ programs, namely confluence and complexity, are discussed in Section 3.4. Section 3.5 shows how to transfer some typical programming patterns used by CHR programmers to the CHR$^{\text{rp}}$ context, and furthermore contains a discussion of alternatives to rule priorities. A basic compilation scheme for CHR$^{\text{rp}}$ programs is given in Section 3.6 and several optimizations are proposed in Section 3.7. The resulting CHR$^{\text{rp}}$ implementation is empirically evaluated in Section 3.8. In Section 3.9 we discuss related work and Section 3.10 concludes the chapter.

3.2 Motivation and Examples

In this section, we show the benefits of our proposed language extension for some typical problems and illustrate them by examples. In these examples, we use CHR on top of Prolog, but the problems apply to other host languages as well.

3.2.1 Constraint Propagators

Constraint solvers generally make use of constraint propagators which filter out inconsistent values from the constraint variables' domains. Efficient solvers use a priority system to make sure that constraint propagators that are computationally cheaper or are expected to have a greater impact, are scheduled early (Ringwelski and Hoche 2005; Schulte and Stuckey 2004). CHR rules are often used as templates for constraint propagators which are instantiated by actual constraints.

Example 3.3 We represent a binary constraint c between two variables x and y as $\mathtt{c}(c, x, y)$. A CHR constraint $\mathtt{d}(x, dx)$ represents that variable x has domain dx. An implementation of two types of constraint propagators for such constraints is given in Listing 3.3.

```
1 :: ac1 @ c(C,X,Y), d(Y,DY) \ d(X,DX0) <=>
                          filter(DX0,C,X,[Y-DY],DX1), DX0 \= DX1 | d(X,DX1).
1 :: ac2 @ c(C,X,Y), d(X,DX) \ d(Y,DY0) <=>
                          filter(DY0,C,Y,[X-DX],DY1), DY0 \= DY1 | d(Y,DY1).

2 :: pc1 @ c(C1,X,Y), c(C2,Y,Z), c(C3,X,Z), d(Y,DY), d(Z,DZ) \ d(X,DX0) <=>
            filter(DX0,(C1,C2,C3),X,[Y-DY,Z-DZ],DX1), DX0 \= DX1 | d(X,DX1).
2 :: pc2 @ c(C1,X,Y), c(C2,Y,Z), c(C3,X,Z), d(X,DX), d(Z,DZ) \ d(Y,DY0) <=>
            filter(DY0,(C1,C2,C3),Y,[X-DX,Z-DZ],DY1), DY0 \= DY1 | d(Y,DY1).
2 :: pc3 @ c(C1,X,Y), c(C2,Y,Z), c(C3,X,Z), d(X,DX), d(Y,DY) \ d(Z,DZ0) <=>
            filter(DZ0,(C1,C2,C3),Z,[X-DX,Y-DY],DZ1), DZ0 \= DZ1 | d(Z,DZ1).

filter(DT0,Cs,T,R,DT1) :-
                 findall(T,(member(T,DT0),once((label(R),call(Cs)))),DT1).

label([X-L|T]) :- member(X,L), label(T).
label([]).
```

Listing 3.3: Arc and path consistency propagators

In this example, the rules $\texttt{ac}_1$ and $\texttt{ac}_2$ implement arc consistency, and rules $\texttt{pc}_1$, $\texttt{pc}_2$ and $\texttt{pc}_3$ implement path consistency. Because the latter is more costly, we assign it a lower priority. The consistencies are implemented by using the `filter`/5 predicate which filters out the inconsistent values in the domain of one of the constraint variables, given the constraints in which it appears and the domains of the other variables involved. The filtering works as follows: for each value in the domain of the target variable (`T`), it is checked whether there exists a labeling for the remaining variables (`R`) such that the constraints on them (`Cs`) hold. The `label`/1 predicate creates a labeling for the variables in its argument, and the built-in predicate `once`/1 ensures only the first successful labeling is considered.[1]

At first glance it might appear that, given the textual order of the rules, the refined operational semantics of CHR ensures that the arc consistency rules are always tried before the path consistency rules. The following situation shows that this is not always the case. Let x be a variable whose domain dx_0 has changed to dx_1 by using one of the arc consistency rules with as active constraint the domain of some variable y. Assume that x is arc consistent. The constraint $\texttt{d}(x, dx_1)$ becomes active and the rules $\texttt{ac}_1$ and $\texttt{ac}_2$ are tried, none of which fires. At that moment, rule $\texttt{pc}_1$ will be tried, while there might still be a variable (e.g. y) that is not arc consistent yet. In contrast, rule priorities ensure that these variables are made arc consistent first. □

[1] In practice, we need to take a copy of the constraints in `Cs` with fresh variables to avoid that the original variables are bound. The code presented here is a simplified version.

3.2.2 Soft Constraints

Bistarelli et al. (2004) propose a framework to deal with *soft constraints* in CHR, based on the *c-semiring* formalism. It works by assigning a score to each possible value of the problem variables, denoting to what extent the soft constraints are satisfied for this value. Hard constraints can be implemented by assigning a score of zero to each value that does not satisfy the constraint. These values can then be removed from consideration. It is useful to process the hard constraints first, as they remove values from the domains of the problem variables, whereas soft constraints generally only update the score of these values. Moreover, constraint propagation for soft constraints requires *combination* of the values of different problem variables, which can be computationally expensive.

In CHR^{rp}, we can assign a high priority to rules implementing hard constraints, a medium priority to rules implementing soft constraints, and finally, a low priority to rules implementing labeling (search). Moreover, we can further differentiate between cheaper and more costly soft constraints, i.e., depending on the number of variables involved.

Example 3.4 We consider variables over finite integer domains, again represented as $\texttt{d}(x, dx)$ constraints, with x a variable and dx a domain, which is a list of pairs $v - s$ with v a value (an integer) and s a score (a floating point number between 0 and 1). The solver of Listing 3.4 implements three constraints, as well as a simple labeling procedure. The first constraint, `odd`/1, is a hard constraint stating that the variable that is its argument, can only take an odd value. The second constraint $\texttt{lt}(x, y, p)$ is a soft constraint that imposes a penalty of p to combinations of values for x and y for which x is not less than y. Finally, there is the $\texttt{all\_different}(x, y, z, p)$ constraint, which is again a soft constraint and which imposes a penalty of p to combinations of values for x, y and z in which these variables do not all have different values. The soft constraints are dealt with by first combining the values of the variables involved, and then projecting the combinations on the different component variables using `project_on_fst`/2, `project_on_snd`/2 and `project_on_trd`/2. The code for the projection predicates is given in Listing 3.5. Basically, it combines the scores for all tuples with the same value for respectively their first, second or third component, using the maximum operation. The `all_different`/4 soft constraint is more expensive to enforce than the `lt`/3 constraint, and so it is given a lower priority. The hard constraint `odd`/1 is given the highest priority, and the labeling rule the lowest. Note the guard in the labeling rule that prevents non-terminating behavior resulting from replacing `d`/2 constraints by identical versions.

□

```
% hard constraint

1 :: odd(X) \ d(X,DX0) <=>
        findall(VX-SX,(member(VX-SX,DX0), VX mod 2 =:= 1),DX1),
        d(X,DX1).

% soft constraint over two variables

2 :: lt(X,Y,P) \ d(X,DX0), d(Y,DY0) <=>
        findall((VX,VY)-S,
            (
                member(VX-SX,DX0), member(VY-SY,DY0),
                (   VX < VY
                ->  S is min(SX,SY)
                ;   S is min(P,min(SX,SY))
                )
            ),Combination),
        project_on_fst(Combination,DX1), d(X,DX1),
        project_on_snd(Combination,DY1), d(Y,DY1).

% soft constraint over three variables

3 :: all_different(X,Y,Z,P) \
    d(X,DX0), d(Y,DY0), d(Z,DZ0) <=>
        findall((VX,VY,VZ)-S,
            (
                member(VX-SX,DX0), member(VY-SY,DY0), member(VZ-SZ,DZ0),
                (   VX \= VY, VX \= VZ, VY \= VZ
                ->  S is min(SX,min(SY,SZ))
                ;   S is min(P,min(SX,min(SY,SZ)))
                )
            ),Combination),
        project_on_fst(Combination,DX1), d(X,DX1),
        project_on_snd(Combination,DY1), d(Y,DY1),
        project_on_trd(Combination,DZ1), d(Z,DZ1).

% labeling

4 :: d(X,DX) <=> DX = [_,_|_] | member(VX-SX,DX), d(X,[VX-SX]).
```

Listing 3.4: Solver for hard and soft constraints

```
project_on_fst(Combination,DX1) :-
        findall(V-S,member((V,_)-S,Combination),L1),
        project(L,DX1).

... % (similar for project_on_snd/2 and project_on_trd/2)

project([V-S0|T],L) :-
        project(T,L0),
        (   select(V-S1,L0,L1)
        ->  S is max(S0,S1),
            L = [V-S|L1]
        ;   L = [V-S0|L0]
        ).
project([],[]).
```

Listing 3.5: Projection for soft constraints

3.2.3 Constraint Store Invariants

It is often desirable to impose certain representational invariants on the constraints in the CHR constraint store. An example of such an invariant is set semantics: no two syntactically equal constraints can exist in the constraint store. These invariants may be violated when asserting new constraints (both built-in and CHR) and we can use special purpose CHR rules for restoring them. Such rules should fire before any rule that expects (a subset of) the invariants.

Example 3.5 Consider that we want to check whether two graphs, G_1 and G_2 are equal. We do this by removing those edges that are common to both graphs. If there are still edges after reaching a fixed point, then the graphs are different. We represent the edges of graph G_1 and G_2 by the edge constraints $\mathtt{e}_1/2$ and $\mathtt{e}_2/2$ respectively. This gives us the program of Listing 3.6.

```
1 :: s1 @ e1(X,Y) \ e1(X,Y) <=> true.
1 :: s2 @ e2(X,Y) \ e2(X,Y) <=> true.

2 :: rc @ e1(X,Y),  e2(X,Y) <=> true.
```

Listing 3.6: Constraint store invariants in $\mathrm{CHR}^{\mathrm{rp}}$

Edges obey set semantics, as implemented by the rules $\mathtt{s}_1$ and $\mathtt{s}_2$. The rule `rc` (**r**emove **c**ommon) removes those edges that appear both in graph G_1 and in graph G_2. Now consider the goal $G = \{\mathtt{e}_1(\mathtt{X},\mathtt{X}), \mathtt{e}_2(\mathtt{X},\mathtt{Y}), \mathtt{e}_2(\mathtt{Y},\mathtt{X}), \mathtt{X} = \mathtt{Y}\}$. Under the refined operational semantics, ignoring the rule priorities, this goal is executed from left to right. Solving the built-in constraint $\mathtt{X} = \mathtt{Y}$ causes the sequential activation

of all three CHR constraints. If the $\mathtt{e}_1/2$ constraint is activated before any of the $\mathtt{e}_2/2$ constraints, then rule `rc` fires before the $\mathtt{s}_2$ rule is tried, which results in a final constraint store containing the $\mathtt{e}_2(\mathtt{X},\mathtt{X})$ constraint. This erroneously indicates that the graphs are different.

We already mentioned in the introduction that rule priorities require that the highest priority rule for which an applicable rule instance exists, fires. This is a global notion in that it does not matter which constraints participate in the firing rule instance, and in particular, there is no concept of an active constraint. So using rule priorities, the set semantics rules will always be tried before the lower priority rule `rc` and when the highest priority applicable rule instance is one of priority 2 (or less), then there will be no two syntactically equal $\mathtt{e}_1/2$ or $\mathtt{e}_2/2$ constraints.

The above example *can* be implemented correctly using the refined semantics, but this leads to inefficient and unreadable code as shown in Listing 3.7. □

```
s1 @           e1(X,Y) \ e1(X,Y) <=> true.
s2 @           e2(X,Y) \ e2(X,Y) <=> true.
s3 @ e1(_,_), e1(X,Y) \ e1(X,Y) <=> true.
s4 @ e1(_,_), e2(X,Y) \ e2(X,Y) <=> true.
s5 @ e2(_,_), e1(X,Y) \ e1(X,Y) <=> true.
s6 @ e2(_,_), e2(X,Y) \ e2(X,Y) <=> true.

rc @ e1(X,Y), e2(X,Y) <=> true.
```

Listing 3.7: Constraint store invariants in regular CHR

3.2.4 Dynamic Rule Priorities

Rule priorities are called *dynamic* if they depend on (the arguments of) the constraints that form a rule instance. Dynamic rule priorities are only known at runtime and different instances of the same rule may have a different priority. In Example 3.2 we have already shown how to implement Dijkstra's shortest path algorithm using dynamic rule priorities. Below, another example program is given.

Example 3.6 (Sudoku) The Sudoku solver from the CHR website (Schrijvers et al. 2008) keeps track of the number of possible values for each Sudoku cell and chooses a value from the most constrained cell first. Listing 3.8 shows the labeling code.

The `f/6` constraints represent a Sudoku cell: the first 4 arguments denote the position of the cell (which 3×3 block and which cell in this block); the 5^{th} argument is the number of remaining possible values for the cell; the 6^{th} argument is a list of these values. If a cell has only one possible value, it is represented by

```
fillone(N), f(A,B,C,D,N,L) <=> member(V,L), f(A,B,C,D,V), fillone(1).
fillone(N) <=> N < 9 | fillone(N+1).
fillone(_) <=> true.
```

Listing 3.8: Labeling rules for a Sudoku solver in CHR

a `f`/5 constraint where the first 4 arguments again denote the position of the cell and the 5th argument is the value of the cell.

Initially, the store contains the constraint `fillone(1)`. The argument of this constraint is increased until a match is found and a rule fires. After the rule has fired, it is reset to 1. In CHR^{rp} we can get the same result using only one rule:

```
N :: f(A,B,C,D,N,L) <=> member(V,L), f(A,B,C,D,V).
```

The remaining rules of the program filter out inconsistent values:

```
1 :: f(A,_,C,_,V) \ f(A,B,C,D,N,L1) <=> select(V,L1,L2) | % same row
                                        N > 1, f(A,B,C,D,N-1,L2).
1 :: f(_,B,_,D,V) \ f(A,B,C,D,N,L1) <=> select(V,L1,L2) | % same column
                                        N > 1, f(A,B,C,D,N-1,L2).
1 :: f(A,B,_,_,V) \ f(A,B,C,D,N,L1) <=> select(V,L1,L2) | % same box
                                        N > 1, f(A,B,C,D,N-1,L2).
```

□

3.3 CHR^{rp}: CHR with Rule Priorities

CHR^{rp} extends CHR with user-defined rule priorities. In this section, we introduce the syntax and semantics of CHR^{rp} and investigate its theoretical properties.

3.3.1 Syntax

The syntax of CHR^{rp} is compatible with the syntax of regular CHR. A CHR^{rp} simpagation rule looks as follows:

$$p :: r \ @ \ H^k \setminus H^r \iff g \mid B$$

where r, H^k, H^r, g and B are as defined in Section 2.2.1. The rule *priority* p is an arithmetic expression for which holds that $\mathsf{vars}(p) \subseteq (\mathsf{vars}(H^k) \cup \mathsf{vars}(H^r))$, i.e., all variables in p also appear in the heads. A rule in which $\mathsf{vars}(p) = \emptyset$ is called a *static* priority rule: its priority is known at compile time and equal for all rule instances. A rule in which $\mathsf{vars}(p) \neq \emptyset$ is called a *dynamic* priority rule: its priority is only known at runtime and different instances of the same rule may fire

1. Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. Apply $\langle \emptyset, H_1 \uplus H_2 \uplus S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle C, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where P contains a rule of priority p of the form $$p :: r \ @ \ H_1' \backslash H_2' \iff g \mid C$$ and a matching substitution θ such that $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, $\theta(p)$ is a ground arithmetic expression and $t = \langle \mathsf{id}(H_1), \mathsf{id}(H_2), r\rangle \notin T$. Furthermore, no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold.

Table 3.1: Transitions of ω_p

at different priorities. In this chapter, all rule priorities are integers or arithmetic expressions that evaluate on integers. In general, rule priorities could be any type of terms for which we have a total preorder.

3.3.2 Operational Semantics

We propose a formal operational semantics for CHR$^{\text{rp}}$. It is called the priority semantics and denoted by ω_p. It consists of a refinement of the ω_t semantics of CHR (see Section 2.2.2) with a minimal amount of determinism in order to support rule priorities. The ω_p semantics uses the same state representation as the ω_t semantics. Its transitions are shown in Table 3.1. The ω_p semantics restricts the applicability of the **Apply** transition with respect to the ω_t semantics. It is only applicable to states with an empty goal and it fires a rule instance of priority p in state σ only if there exists no ω_t **Apply** transition $\sigma \overset{\omega_t}{\mapsto}_P \sigma'$ that fires a rule instance of a higher priority. The **Solve** and **Introduce** transitions are unchanged. The ω_t derivation for the `leq` program in Example 2.2 is also a valid ω_p derivation.

We illustrate the differences between the ω_p and ω_r semantics on some small examples. The first example shows that rule priorities are different from rule order under the ω_r semantics.

Example 3.7 Consider the following program:

```
1 :: r1 @    a ==> b.
2 :: r2 @ a, b ==> c.
3 :: r3 @    a <=> true.
4 :: r4 @ a, b ==> d.
```

Using the refined operational semantics, the rule priority declarations are ignored. For the initial goal $G = \{\mathtt{a}\}$, we get the following results. Under ω_p, the unique

qualified answer for goal G is $\{\mathtt{b}, \mathtt{c}\}$ whereas under the ω_r semantics, the answer is $\{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The difference is explained as follows: under the ω_p semantics, constraint `a` is removed by rule $\mathtt{r}_3$ before rule $\mathtt{r}_4$ is tried. This causes rule $\mathtt{r}_4$ to be not applicable anymore. There is no rule ordering that can cause rule $\mathtt{r}_3$ to be fired after rule $\mathtt{r}_2$ but before rule $\mathtt{r}_4$ in the ω_r semantics. □

The following two examples illustrate the non-determinism in the ω_p semantics. Note that this non-determinism could be removed by further refining the ω_p semantics (e.g., using rule order, recency, etc. to resolve conflicts). However, we prefer to keep all the determinism users can rely on, explicit in the priority annotations.

Example 3.8 Consider the following program:

```
1 :: r1 @ a(X) ==> write(r1:X), nl.
2 :: r2 @ a(X) ==> write(r2:X), nl.
```

Note that this program produces side effects and as such is not pure. For the initial goal $\{\mathtt{a}(1), \mathtt{a}(2)\}$, we get the following output:

Output in ω_r	Alternative Outputs in ω_p			
$\mathtt{r}_1 : 1$	$\mathtt{r}_1 : 1$	$\mathtt{r}_1 : 1$	$\mathtt{r}_1 : 2$	$\mathtt{r}_1 : 2$
$\mathtt{r}_2 : 1$	$\mathtt{r}_1 : 2$	$\mathtt{r}_1 : 2$	$\mathtt{r}_1 : 1$	$\mathtt{r}_1 : 1$
$\mathtt{r}_1 : 2$	$\mathtt{r}_2 : 1$	$\mathtt{r}_2 : 2$	$\mathtt{r}_2 : 1$	$\mathtt{r}_2 : 2$
$\mathtt{r}_2 : 2$	$\mathtt{r}_2 : 2$	$\mathtt{r}_2 : 1$	$\mathtt{r}_2 : 2$	$\mathtt{r}_2 : 1$

Here the non-determinism is caused by the existence of different rule instances for the same rule. □

Example 3.9 Consider the following program:

```
1 :: r1 @ a ==> write('rule 1'), nl.
1 :: r2 @ a ==> write('rule 2'), nl.
```

In this example, rules $\mathtt{r}_1$ and $\mathtt{r}_2$ have an equal priority. For the initial goal `a`, we get the following output:

Output in ω_r	Alt. Outputs in ω_p	
`rule 1`	`rule 1`	`rule 2`
`rule 2`	`rule 2`	`rule 1`

Here, the non-determinism is caused by the existence of different rules with equal priority. □

A final example compares the ω_p semantics with the ω_t semantics and shows that CHR^{rp} programs are not always monotonic.

Example 3.10 A form of negation by absence (see also (Van Weert et al. 2006)) can be implemented in CHR$^{\text{rp}}$ as follows:

```
1 :: r1 @ a \ no_a <=> fail.
2 :: r2 @     no_a <=> true.
```

Under the ω_t semantics, the goal $\{\texttt{a}, \texttt{no\_a}\}$ either fails or succeeds with qualified answer $\{\texttt{a}\}$, whereas under the ω_p semantics it must fail and in particular, the second rule cannot fire. The goal $\{\texttt{no\_a}\}$ succeeds under both semantics by means of firing rule $\texttt{r}_2$. □

3.3.3 Correspondence between ω_p and ω_t Derivations

In this subsection, we show the correspondence between the ω_p semantics of CHR$^{\text{rp}}$ and the ω_t semantics of CHR. Every CHR$^{\text{rp}}$ program is a CHR program if we ignore the priority annotations. We show that every ω_p derivation is also a derivation under ω_t (Theorem 1). We then prove that the ω_p semantics respects rule priorities (Theorem 2). Finally, for CHR$^{\text{rp}}$ programs in which all rule priorities are equal, we show that every ω_t derivation corresponds to an ω_p derivation (Theorem 3).

Theorem 1 *Every derivation D under ω_p, is also a derivation under ω_t. If a state σ is a final state under ω_p, then it is also a final state under ω_t.*

Proof: The first part of the theorem holds because ω_p only adds restrictions to the applicability of ω_t transitions. For the second part, suppose that state σ is a final state under ω_p, but not under ω_t. The only transition applicable under ω_t must be the **Apply** transition, since the **Solve** and **Introduce** transitions are equal in both semantics. This means that the goal must be empty.

From all **Apply** transitions that are applicable in state σ under ω_t, we can choose the one that fires the highest priority rule instance. It is clear that this transition is also applicable under ω_p, which contradicts our assumption. This proves the second part of the theorem. □

Theorem 2 *If an **Apply** transition is applied to a state σ under ω_p, firing a rule instance of priority p, there exists no derivation under ω_t and starting in σ in which the first **Apply** transition fires a higher priority rule instance.*

Proof: The ω_p **Apply** transition, applied on state σ, fires the highest priority rule instance that can fire given the current built-in store, CHR store and propagation history. If there exists an ω_t derivation D starting in σ in which the first **Apply** transition fires a rule instance of a higher priority, then D must contain a **Solve** or **Introduce** transition that updates respectively the built-in store or CHR store, and that makes the rule instance applicable. Since the ω_p **Apply** transition requires the goal to be empty, no such derivation can exist. □

For every state $\sigma = \langle G, S, B, T\rangle_n$, there exists a derivation $\sigma \overset{\omega_p}{\longmapsto}{}^*_P\ \sigma^*$ where $\sigma^* = \langle \emptyset, S \cup S', B \wedge B', T\rangle_{n+|S'|}$ and $G = B' \uplus \mathsf{chr}(S')$, B' is a multi-set of built-in constraints, S' is a set of identified CHR constraints and $|S'|$ is the number of elements in S'. The derivation is formed by solving all built-in constraints, and introducing all CHR constraints in the goal G. We call state σ^* a *normalization* of σ. There are $|S'|!$ such normalizations, one for each order in which the CHR constraints of the goal are introduced.

Theorem 3 *For a given CHR^{rp} program P in which all rule priorities are equal, it holds that for every non-failing derivation D under ω_t, if $\sigma_1 \overset{\omega_t}{\longmapsto}_P \sigma_2 \in D$ then for every normalization σ_2^* of σ_2, there exists a normalization σ_1^* of σ_1 such that $\sigma_1^* \overset{\omega_p}{\longmapsto}{}^*_P\ \sigma_2^*$. If a state σ is a final state under ω_t, it is also a final state under ω_p.*

Proof: Given $\sigma_1 \overset{\omega_t}{\longmapsto}_P \sigma_2 \in D$. We look at each of the three possible transitions:

1. **Solve** $\sigma_1 = \langle \{c\} \uplus G, S, B, T\rangle_n$ and $\sigma_2 = \langle G, S, c \wedge B, T\rangle_n$. Clearly all normalizations of σ_1 and σ_2 are equal.

2. **Introduce** $\sigma_1 = \langle \{c\} \uplus G, S, B, T\rangle_n$ and $\sigma_2 = \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$. All normalizations of σ_2 are also normalizations of σ_1.

3. **Apply** $\sigma_1 = \langle G, H^r \uplus S, B, T\rangle_n$ and $\sigma_2 = \langle C \uplus G, S, \theta \wedge B, T'\rangle_n$. In any normalization of σ_1, the same rule instance can fire because introducing CHR constraints to the store nor solving built-in constraints from the goal can prevent a rule instance from being applicable.[2] So we have for every normalization σ_1^* of σ_1 that $\sigma_1^* = \langle \emptyset, H^r \cup S \cup S', B \wedge B', T\rangle_{n'} \overset{\omega_p}{\longmapsto}_P \langle C, S \cup S', \theta \wedge B \wedge B', T'\rangle_{n'} = \sigma_2'$. It is easy to see that every normalization of σ_2 corresponds to a normalization of such a state σ_2' for some normalization of σ_1.

We conclude that for every transition $\sigma_1 \overset{\omega_t}{\longmapsto}_P \sigma_2$, there exists a corresponding derivation $\sigma_1^* \overset{\omega_p}{\longmapsto}_P \sigma_2^*$ for every normalization σ_2^* of σ_2. The second part of the theorem follows from Theorem 1. □

Theorem 3 implies that for CHR$^{\mathrm{rp}}$ programs in which all rule priorities are equal, every execution strategy under ω_t is consistent with ω_p, and so such programs can be executed using the refined operational semantics as implemented by current CHR implementations. While such CHR$^{\mathrm{rp}}$ programs are obviously degenerate, we can retain many advantageous aspects of the refined operational semantics of CHR when compiling CHR$^{\mathrm{rp}}$ programs (see Section 3.6).

[2] We do not consider non-monotone guards like Prolog's `var`/1. They are not allowed in pure CHR.

3.4 Program Properties

In this section, we investigate two important properties of CHR$^{\text{rp}}$ programs, namely confluence and complexity.

3.4.1 Confluence

Confluence is the property that a program always gives the same answers for a given goal, regardless of the execution path followed. In Section 2.6.1, we showed that, for terminating programs, confluence under the theoretical operational semantics of CHR can be decided by checking whether all *critical pair* states are joinable. A critical pair consists of two *minimal* states σ_1 and σ_2 that result from a common ancestor state σ after firing different rule instances, such that the rule instance that led to σ_1 cannot fire in σ_2 and vice versa. Two states are joinable if they both derive into states that are *variants* of each other. More formal definitions can be found in Section 2.6.1 and in (Abdennadher 1997). Clearly, under the ω_p semantics, a critical pair only follows from firing rule instances with equal priority.

If a rule instance $\theta(r)$ is applicable in state $\langle G, S, B, T\rangle_n$ under ω_t, then this is also the case in any (non-failed) 'larger' state $\langle G \uplus G', S \cup S', B \wedge B', T \setminus T'\rangle_{n'}$. This result does not hold under ω_p because adding CHR or built-in constraints to the store or removing propagation history tuples, can cause a *higher priority* rule instance to become applicable. A similar problem was found by Duck (2005) for the refined operational semantics.

Example 3.11 Consider the following example, adapted from (Duck 2005) and extended with rule priorities:

```
1 :: r1 @ p, q(_) <=> r.
2 :: r2 @ q(_), q(_) \ r <=> true.
3 :: r3 @ r \ q(_) <=> true.
4 :: r4 @ r <=> true.
```

A critical pair for rule $\mathtt{r_1}$ is

$$(\langle\{\mathtt{r}\}, \{\mathtt{q(1)\#1}\}, \mathtt{true}, T_1\rangle_n, \langle\{\mathtt{r}\}, \{\mathtt{q(2)\#2}\}, \mathtt{true}, T_2\rangle_n)$$

with common ancestor state

$$\langle\emptyset, \{\mathtt{q(1)\#1}, \mathtt{q(2)\#2}, \mathtt{p\#3}\}, \mathtt{true}, T\rangle_n$$

Both states further derive into $\langle\emptyset, \emptyset, \mathtt{true}, T'\rangle_{n+1}$ and so it seems that there is confluence. However, given the initial goal $\{\mathtt{p}, \mathtt{q(1)}, \mathtt{q(2)}, \mathtt{q(3)}\}$ we get the qualified answers $\{\mathtt{q(1)}, \mathtt{q(2)}\}$, $\{\mathtt{q(2)}, \mathtt{q(3)}\}$ or $\{\mathtt{q(1)}, \mathtt{q(3)}\}$. The problem is that in a minimal state, rule $\mathtt{r_2}$ is not applicable and rules $\mathtt{r_3}$ and $\mathtt{r_4}$ can fire. In a larger state, $\mathtt{r_2}$ may

become applicable and cause rules $\mathtt{r_3}$ and $\mathtt{r_4}$ to be not applicable anymore. This example shows that the fact "all critical pairs are joinable" does not necessarily imply local confluence (and hence confluence) under CHR^{rp}. □

However, since every ω_p derivation is a valid ω_t derivation, we can use confluence w.r.t. ω_t to prove confluence w.r.t. ω_p.

Corollary 1 *If program P is terminating under ω_p, and confluent under ω_t, then it is also confluent under ω_p.*

Proof: The proof is by contradiction. Assume that there exists a program P which is terminating under ω_p, confluent under ω_t but not confluent under ω_p. In that case, there exists a state σ such that $\sigma \overset{\omega_p}{\longmapsto}{}^*_P \sigma_1$ and $\sigma \overset{\omega_p}{\longmapsto}{}^*_P \sigma_2$ with σ_1 and σ_2 final ω_p states that are not variants of each other. By Theorem 1, we then have that $\sigma \overset{\omega_t}{\longmapsto}{}^*_P \sigma_1$ and $\sigma \overset{\omega_t}{\longmapsto}{}^*_P \sigma_2$, with σ_1 and σ_2 final ω_t states. Since σ_1 and σ_2 are not variants, P is not confluent under ω_t, which contradicts our assumption. □

If a program P passes the standard ω_t confluence test (by ignoring rule priorities), then P is also confluent under ω_p.

Example 3.12 Consider the `leq` program from Example 2.1. This program is confluent under ω_t. Therefore, by Corollary 1, the `leq` program is confluent under ω_p. □

Note that the converse is not true, that is, there exist programs that are confluent w.r.t. the ω_p semantics, but not w.r.t. the ω_t semantics. For example, consider the following program.

```
1 :: p <=> q.
2 :: p <=> r.
```

Clearly the program is non-confluent under ω_t. However, under ω_p the program is confluent because the first rule is always preferred. In general, requiring a program to be confluent under ω_t is too restrictive, so we shall consider some alternative ideas.

Practical Confluence Test

For the refined operational semantics of (regular) CHR, we can fall back on a *practical test* to help decide confluence. This test works by examining the non-determinism of each of the ω_r transitions w.r.t. the given program P. In particular, there are two sources of non-determinism: the order in which constraints are reactivated by the **Solve** transition is not determined, and neither is the order in which rule matches are found by the **Simplify** and **Propagate** transitions. In (Duck 2005), the non-determinism resulting from the **Solve** transition is eliminated by requiring that this transition never reactivates any constraints (*trivial wake-up*

policy). Under this condition, one only needs to consider those rule instances that fire with the same active occurrence.

Now, the practical confluence test consists of proving that all occurrences are *matching complete* or *matching independent* and that the matching complete occurrences are *order independent*. The exact definitions are given in (Duck 2005, Chapter 6), we just give the intuition here. Matching completeness means that all rule instances involving a given active occurrence, eventually get to fire, i.e., none of their constituent constraints are (directly or indirectly) removed. For example, an active `a`/0 occurrence matching the left-most head in the rule

```
a \ b(X) <=> c(X).
```

is matching complete given that the `c`/1 constraint does not (directly or indirectly) remove `a`/0 or `b`/1 constraints.

Matching completeness in itself is not sufficient as firing a rule instance may still affect parts of the constraint store, without invalidating the other matches. Order independence is a criterion that ensures this is not a problem. For example, let there also be a rule

```
c(X), c_list(L) <=> c_list([X|L]).
```

then the result is dependent on the order in which `b`/1 constraints are converted into `c`/1 constraints. On the other hand, the rule

```
c(X), c_sum(S) <=> c_sum(X+S).
```

is independent of the order, as addition is an associative and commutative operator.

Matching independence means that the result is independent of the rule instance that fired. For example, an active `b`/0 occurrence, matching the right-most head in the rule

```
a(_) \ b <=> c.
```

is matching independent, as the result does not depend on the argument of the `a`/1 constraint, and hence not on the exact rule instance that fired.

In the ω_p semantics of CHR$^{\text{rp}}$, the only relevant source of non-determinism is in the **Apply** transition.[3] However, unlike under the ω_r semantics, different *rules* may fire in a given state.[4] Moreover, there does not need to be a constraint that participates in all rule instances that are applicable. Therefore, in general, we need to consider more cases than under the refined semantics. Still, the concepts used for the refined confluence test, transfer to the ω_p case. For example, consider the following rules, adapted from the ray tracer program of Duck (2005).

[3] The **Solve** and **Introduce** transitions also introduce non-determinism, but this has no effect on confluence.

[4] These rules do need to have the same priority though.

```
1 :: sphere(I,X3,Y3,Z3,R,_) \ light_ray(X1,Y1,Z1,X2,Y2,Z2,_,J) <=> I \= J,
        sphere_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3,R,U1,U2),
        sphere_blocks(U1,U2) | true.
1 :: plane(I,A,B,C,D,_) \ light_ray(X1,Y1,Z1,X2,Y2,Z2,_,J) <=> I \= J,
        plane_intersection_calculation(X1,Y1,Z1,X2,Y2,Z2,A,B,C,D,U),
        plane_blocks(U) | true.
```

The rules remove light rays that are blocked by either a sphere or a plane. Clearly, all light rays that are blocked, are eventually removed by these rules, and it does not matter which rule instance removes a given light ray when it is blocked by multiple objects. So for these rules, we have a combination of matching completeness and matching independence. More precisely, we can partition the set of all rule instances into subsets of matching independent instances (those involving the same `light_ray`/8 constraint) such that each rule firing invalidates only those rule instances that belong to the same subset. That said, we leave a practical confluence test for the ω_p semantics as future work.

Observable Confluence

Finally, we link confluence under the ω_p semantics to observable confluence as described in (Duck et al. 2007). Observable confluence is based on the observation that certain critical pair states cannot occur in practice because they are not reachable from any valid goal. Therefore, Duck et al. (2007) introduce $\mathcal{I}$-confluence: confluence with respect to some invariant $\mathcal{I}$. Invariants are properties of states and are often restrictions on combinations of constraints that may occur in the constraint store. For example, for the union-find program of Schrijvers and Frühwirth (2006), one invariant is that there are no two `find`/2 constraints with the same first argument. We can express this invariant by the rule

```
1 :: find(X,_), find(X,_) ==> fail.
```

By giving all other rules of the original program a priority of 2, we have that confluence under the ω_p semantics implies $\mathcal{I}$-confluence under ω_t for the given invariant. Indeed, only critical pair states exist that satisfy the invariant because if not, their ancestor state would deterministically fail.

3.4.2 Complexity

In Chapter 5, a meta-complexity result for CHR$^{\text{rp}}$ is given that allows one to derive the time complexity of a CHR$^{\text{rp}}$ program by reasoning amongst others about the number of rule applications. The approach is based on the Logical Algorithms formalism by Ganzinger and McAllester (2002) and relies on an optimized implementation of CHR$^{\text{rp}}$. The result of Sneyers et al. (2005, 2008) that states that any algorithm can be implemented in CHR with optimal time and space complexity, also easily transfers to the CHR$^{\text{rp}}$ context. However, it requires memory

optimizations that are currently not implemented by our CHR^{rp} compiler (see (Sneyers et al. 2006b)). Noteworthy is the problem of memory initialization in the RAM machine simulator program presented by Sneyers et al. (2005). In that program, the RAM machine's memory cells are represented as CHR constraints of the form $\mathtt{m}(a, c)$ with a an address and c the content of the memory cell at that address. This content is updated by rules like the one below.

```
prog(L,L1,const,B,A) \ m(A,_), pc(L) <=> m(A,B), pc(L1).
```

The rule states that if the current instruction is a `const` instruction with arguments b and a, the content of the memory cell with address a is replaced by b. The RAM simulator program expects the existence of an $\mathtt{m}/2$ constraint for each memory address that can be referred to. However, in CHR^{rp} we can relax this requirement by adding a second rule

```
prog(L,L1,const,B,A) \ pc(L) <=> m(A,B), pc(L1).
```

and assigning it a lower priority than the first one. This second rule will then create a new $\mathtt{m}/2$ constraint if no such constraint exists for a given address a. The same approach works in regular CHR, but only if we rely on the refined operational semantics.

3.5 Discussion

In this section, we show how some typical programming patterns CHR programmers use (relying on the refined operational semantics of CHR) transfer to CHR^{rp}. Furthermore, we look at some alternative ways of expressing execution control in CHR, and relate them to rule priorities.

3.5.1 Programming Patterns

In this subsection, we show that certain programming patterns that are often used in CHR programs based on the refined operational semantics, lead to problems when using the priority semantics. We show how alternative formulations lead to the desired result using rule priorities.

Set Semantics and the Propagation History

Consider the following CHR^{rp} program:

```
1 :: r1 @ a \ a <=> true.
2 :: r2 @ a ==> a.
```

and the initial goal $G = \{\texttt{a}\}$. Under the refined operational semantics, ignoring rule priorities, this program terminates after two rule firings: first rule $\texttt{r}_2$ fires, adding a new constraint `a`, which is then removed by rule $\texttt{r}_1$. Under the priority semantics, it is possible that not the newly added instance of `a` is removed, but instead the one that was in the initial goal. The new instance has no propagation history tuple for rule $\texttt{r}_2$ and so the rule can fire again. If rule $\texttt{r}_1$ always removes the oldest instance of `a`, the program does not terminate.

We can solve this problem by introducing a new constraint `new_a` that is removed if a constraint `a` already exists, and replaced by a new `a` constraint otherwise. The following program terminates under the priority semantics for the initial goal $G = \{\texttt{new\_a}\}$.

```
1 :: r4 @ a \ new_a <=> true.
2 :: r5 @ new_a <=> a.
3 :: r6 @ a ==> new_a.
```

Non-ground constraints can be made syntactically equal by unification. In this case, the above approach does not work, but on the other hand, the refined operational semantics also cannot guarantee termination. For example, the CHR program

```
r7 @ a(X,_) \ a(X,_) <=> true.
r8 @ a(X,f) ==> a(Y,F), X = Y, F = f.
```

does not terminate in the (ω_r based) K.U.Leuven CHR system (Schrijvers and Demoen 2004) for goal $G = \{\texttt{a}(\texttt{X}, \texttt{f})\}$.

CHR Constraints in Guards

Although in theory not allowed, many useful programs use CHR constraints in the guards of rules. In particular, this is often done to detect the absence of a constraint, as in the following example:

```
r1 @ a(X) \ b(Y) <=> no_c(X) | d(X,Y).
r2 @ c(X) \ no_c(X) <=> fail.
r3 @ no_c(_) <=> true.
```

The `no_c` constraint acts as an ask constraint. In particular, if asserting it succeeds, neither the built-in constraint store, nor the CHR constraint store are changed. In CHR implementations based on the refined operational semantics, such a constraint is solved for immediately, and so it may work in practice.[5] It is not immediately clear how a CHR^{rp} implementation can support CHR constraints in the

[5]Some optimizations may lead to unexpected behavior though. For example, late storage may have as effect that some constraints are not observed the moment a CHR constraint in the guard is solved.

guard. In particular, problems may arise when these constraints participate in rules with a lower priority than the one they are guarding.

The following CHR^{rp} code essentially does the same as the CHR code above, but without using CHR constraints in the guard.

```
1 :: r4 @ b(X) <=> b_id(X,_).
3 :: r5 @ a(X), b_id(Y,Id) ==> r1_instance(X,Y,Id).
1 :: r6 @ c(X) \ r1_instance(X,_,_) <=> true.
2 :: r7 @ r1_instance(X,Y,Id), b(_,Id) <=> d(X,Y).
```

We add a unique identifier argument (e.g., a fresh variable) to the removed constraints of the original rule $\mathtt{r}_1$. This is because only if the rule fires, these constraints are removed and the decision to fire or not depends on whether rule $\mathtt{r}_6$ or $\mathtt{r}_7$ fires. Consider that we do not use identifiers, but instead remove a syntactically equal constraint as shown below.

```
3 :: r8 @ a(X), b(Y) ==> r1_instance(X,Y).
1 :: r9 @ c(X) \ r1_instance(X,_) <=> true.
2 :: r10 @ r1_instance(X,Y), b(Y) <=> d(X,Y).
```

Now rule $\mathtt{r}_{10}$ can remove a constraint $\mathtt{b}/1$ for which rule $\mathtt{r}_8$ does not have a propagation history tuple, while leaving a syntactically equal $\mathtt{b}/1$ constraint in the store for which such a tuple does exist. With an initial goal $\{\mathtt{a}(1), \mathtt{b}(2), \mathtt{b}(2)\}$ it depends on the matching order whether one or both $\mathtt{b}/1$ constraints are removed.

We note that neither rule $\mathtt{r}_1$ nor rule $\mathtt{r}_5$ 'triggers' when a $\mathtt{c}/1$ constraint is removed and so only a passive form of negation as absence is supported. For more on negation as absence in CHR, see (Van Weert et al. 2006).

Sequential Host Language Statements

Rule bodies often contain host language statements that are not constraints. Amongst others, this is the case for IO calls. Another example is the Prolog `is/2` predicate which throws an exception if the second argument (right hand side) is not ground. Under the refined operational semantics, rule bodies are processed from left to right and all constraints are solved for as soon as they are encountered. In contrast, under the priority semantics, the order in which different constraints in a rule body are processed is not determined and CHR constraints are only *introduced* into the CHR constraint store (i.e., they are not activated immediately).[6]

We can ensure correct behavior of these host language statements by encapsulating them into CHR constraints. Consider the following rule:

```
r1 @ a(X) <=> write('Give a number'), read(Y), Z is X - Y, b(Z).
```

[6] In our implementation, bodies are processed from left to right, so host language statements are executed in the correct order. However, in theory, there are no guarantees.

The `write`/1 statement should be executed before the `read`/1 statement and the call to `is`/2 should be delayed until `X - Y` is ground.

```
3 :: r2 @ a(X) <=> io(yes,write('Give a number'),Next),
                   io(Next,read(Y),_), safe_is(Z,X - Y), b(Z).
1 :: r3 @ io(yes,Call,Done) <=> call(Call), Done = yes.
1 :: r4 @ safe_is(X,Y) <=> ground(Y) | X is Y.
```

Each IO call *call* is encapsulated into a `io`(*ready*, *call*, *done*) constraint where *ready* equals `yes` if the IO call is ready to be executed, and *done* equals `yes` if the IO call is done executing. Rule $\mathtt{r}_3$ fires if an encapsulated IO call is ready to be executed (*ready* = `yes`). Its body contains the encapsulated call and instantiates *done* to `yes`. This allows the next IO call to be executed. Note that the ω_p semantics does not determine the order in which constraints from the body are solved or introduced, and so the order in which these constraints appear in the body is irrelevant. However, ω_p does state that all body constraints are processed before the next rule firing. The `is`/2 call is encapsulated in a `safe_is`/2 constraint. It is called by rule $\mathtt{r}_4$ as soon as its second argument is ground.

3.5.2 Alternatives for Rule Priorities

In this subsection, we discuss some alternatives for rule priorities that also allow for execution control. First, we look at how rule priorities can be extended to support priority aging.

Starvation and Priority Aging

Under the priority semantics, a given rule instance does not fire as long as there exists a higher priority applicable rule instance. To prevent that a rule instance has to 'wait' very long, or even never gets to fire ('starvation'), we can increase its priority over time. In operating systems this is called priority aging.

We can extend the rule priority declaration with a function that updates the actual priority of a given rule instance, and a declaration of how often this update should happen (expressed as number of rule firings). For example

```
⟨X,-1,10⟩ :: r1 @ a(X) <=> b(X).
```

would then mean that an instance of rule $\mathtt{r}_1$ initially has priority `X` and this priority 'increases' by one every 10 rule firings.

The main difficulty with this approach is determining when a rule instance becomes applicable. This seems to be only possible if we use an eager matching technique like in the RETE algorithm (Forgy 1982), where all applicable rule instances are eagerly generated and kept in memory. This approach is however

expensive as far as memory consumption is concerned, and because firing a rule instance may invalidate other ones. Also, traversing all rule instances to update their priority, including the cost of updating the agenda, can be expensive.[7]

We can emulate the proposed extension of rule priorities by using a source transformation. Consider the following, somewhat more complex rule:

```
⟨X,-1,10⟩ :: r2 @ a(X) \ b <=> X mod 2 =:= 1 | c(X).
```

We add a unique identifier as extra argument to all CHR constraints and transform the rule as follows:

```
  1 :: r3 @ a(X,Id1), b(Id2) ==> X mod 2 =:= 1 | r2_instance(X,1,Id1,Id2).
  1 :: r4 @ update \ r2_instance(Prio,10,Id1,Id2) <=>
            r2_instance_passive(Prio-1,1,Id1,Id2).
  1 :: r5 @ update \ r2_instance(Prio,Step,Id1,Id2) <=> Step < 10 |
            r2_instance_passive(Prio,Step+1,Id1,Id2).
  2 :: r6 @ update <=> true.
  3 :: r7 @ r2_instance_passive(Prio,Step,Id1,Id2) <=>
            r2_instance(Prio,Step,Id1,Id2).
P+3 :: r8 @ a(X,Id1) \ b(Id2), r2_instance(P,Step,Id1,Id2) <=>
            c(X,_), update.
P+4 :: r9 @ r2_instance(P,_,_,_) <=> true.
```

Rule `r3` fires when an applicable rule instance is found. It generates a constraint representing this instance with its initial priority and a step count that denotes how many rule firings have taken place since the last update of its priority. Rule instance priorities are updated after each rule firing by the `update` constraint. This constraint is added to the body of each rule (see rule `r8`). Rule `r4` increases the priority of a rule instance of rule `r2` whenever the step count equals 10. Rule `r5` increases the step count by one if it is less than 10. Both rules `r4` and `r5` generate a *passive* version of the rule instance to ensure that each instance is updated only once after every rule firing. Rule `r6` removes the `update` constraint after all rule instances have been updated and rule `r7` converts the passive versions of the rule instances back into active versions. Rule `r8` fires a rule instance if it has the highest priority and all head constraints are still in the store and rule `r9` removes a rule instance if some of the head constraints are not in the store anymore.

The above approach assumes that guards are monotone: once a guard is implied by the built-in constraint store, it remains so in all later states.

[7] See Section 3.6 for details on the compilation of CHR^{rp} programs; when using Fibonacci heaps (Fredman and Tarjan 1987) to implement the agenda, the (amortized) cost of increasing the priority of an item is constant.

Constraint Priorities

CHR constraints can often be divided into (active) *operation* constraints and (passive) *data* constraints (see for example the CHR implementation of the union-find algorithm by Schrijvers and Frühwirth (2006) and Fibonacci heaps by Sneyers et al. (2006a)). This is in particular the case when using CHR as a general purpose programming language. Often such programs do not have a declarative reading in classical logic, but do have one in intuitionistic linear logic (Betz and Frühwirth 2005).

By using constraint priorities, we can prioritize certain operation constraints and hence certain operations. The priority of a constraint depends on its type and potentially also on its arguments. An obvious semantics for constraint priorities is the following: if in a given execution state, c is the highest priority constraint for which an applicable rule instance exists, then this constraint (or a constraint with equal priority) must participate in the next rule firing.

In the following rule, let $p_1, \ldots, p_n$ be the constraint priorities of constraints $c_1(\bar{X}_1), \ldots, c_n(\bar{X}_n)$ respectively and let lower numbers denote higher priorities.

$$c_1(\bar{X}_1), \ldots, c_n(\bar{X}_n) \iff g \mid B$$

We can use rule priorities to get the same execution strategy as proposed for constraint priorities:

$$\min(p_1, \ldots, p_n) :: c_1(\bar{X}_1), \ldots, c_n(\bar{X}_n) \iff g \mid B$$

Clearly, constraint priorities are subsumed by rule priorities. The opposite does not hold, as the following simple example illustrates.

```
1 :: r1 @ a(X) <=> b(X).
X :: r2 @ a(X) <=> c(X).
```

We can of course add an extra head to each rule, whose constraint priority equals the rule priority, as shown below:

```
r3 @ r3_priority \ a(X) <=> b(X).
r4 @ r4_priority(X) \ a(X) <=> c(X).
```

where the constraint `r3_priority` has priority 1 and the constraint `r4_priority`(x) has priority x. However, these constraints have to be asserted first, which is feasible for static priority rule `r3`, but not for dynamic priority rule `r4` as there are infinitely many `r4_priority`/1 constraints. Therefore, we conclude that constraint priorities subsume static rule priorities, but not dynamic rule priorities.

The Extended Constraint Handling Rules library (`ech`) of the ECL^{i}PS^e Constraint Logic Programming system (Wallace et al. 1997) supports a form of static

constraint priorities. Operationally, it is based on the concept of an *active* constraint as in the refined semantics. If a new CHR constraint is asserted in the body of a rule, it becomes active if it has a higher priority than the current active constraint, and otherwise it is scheduled for activation at its own (lower) priority. If a built-in constraint wakes up a set of CHR constraints, then these are activated from highest to lowest priority as long as their priority is higher than that of the current active constraint, and scheduled for activation otherwise. In `ech`, a CHR constraint has a priority between 1 and 11. These priorities can be specified absolutely, or relative to the priority of the CHR solver (which is 9 by default). The priority system is shared with other constraint solver libraries.

Example 3.13 Listing 3.9 shows an example program with constraint priorities as supported by the `ech` library.

```
:- constraints a/1:at_absolute_priority(1),
               b/0:at_absolute_priority(2),
               c/1:at_absolute_priority(3),
               d/0:at_absolute_priority(4),
               e/1:at_absolute_priority(5).

r1 @ d ==> c(X), a(X), e(X), X = 1, b.

r2 @ a(1) <=> write(a).
r3 @ b    <=> write(b).
r4 @ c(1) <=> write(c).
r5 @ d    <=> write(d).
r6 @ e(1) <=> write(e).
```

Listing 3.9: CHR program with constraint priorities in `ech`

Consider the goal {`d`}. Constraint `d` becomes active and fires rule $\mathtt{r}_1$. Constraints $\mathtt{c}(x)$, $\mathtt{a}(x)$, and $\mathtt{e}(x)$ (for some fresh variable x) are all (sequentially) inserted into the constraint store. Of these constraints, $\mathtt{c}(x)$ and $\mathtt{a}(x)$ are activated as soon as they are asserted, whereas $\mathtt{e}(x)$ is scheduled for activation at priority 5. The built-in constraint $x = \mathtt{1}$ wakes up $\mathtt{a}(1)$, $\mathtt{c}(1)$ and $\mathtt{e}(1)$ in priority order: $\mathtt{a}(1)$ fires rule $\mathtt{r}_2$ and is then removed; $\mathtt{c}(1)$ fires rule $\mathtt{r}_4$ and is removed; $\mathtt{e}(1)$ is (again) scheduled for activation at priority 5. Then `b` is asserted and becomes active immediately. It fires rule $\mathtt{r}_3$ and is removed. Now the body of rule $\mathtt{r}_1$ has been processed completely and constraint `d` searches for the next applicable rule instance. This next rule instance is an instance of $\mathtt{r}_5$. It fires and `d` is removed. Finally, $\mathtt{e}(1)$ is activated, fires rule $\mathtt{r}_6$ and is removed. The (screen) output of this query hence is "`acbde`".

The constraint priorities behave somewhat illogically since a constraint becomes active as soon as it has a higher priority than the current active constraint,

Current Phase	Rule Name	Next Phase
`start`	`assign_first_seat`	`assign_seats`
`assign_seats`	`find_seating`	`extend_path`
`extend_path` `extend_path`	`extend_path` (*default*)	`extend_path` `check_done`
`check_done` `check_done`	`are_we_done` (*default*)	`print_results` `assign_seats`
`print_results`	`print_results`	`print_results`

Table 3.2: Phase transitions for Miss Manners

while there might be higher priority constraints in the remaining part of the rule body. If rule bodies were processed completely before activating a higher priority constraint, the example program would output "`abcde`". □

Execution in Phases

Several rule based algorithms partition their rules into phases. Only the rules of a single phase called the *current* phase, are allowed to fire and the current phase changes either after firing a rule, or when no rule is applicable in it. A similar idea is used in term rewriting systems. In TAMPR (Boyle et al. 1997), a set of rewrite rules is partitioned into a sequence of subsets of these rules. All rules in a given subset are applied until reaching a fixpoint, after which the same is done for the next subset. Stratego (Visser 2001) uses a similar system where rewrite rules are divided into *stages*. A single rule can belong to multiple stages.

Example 3.14 (Miss Manners) The Manners program is a well known benchmark for production rule systems (Brant et al. 1991). It assigns a set of people seats on a round table such that people sitting next to each other are of opposite sex, and every person shares a hobby with its left or right neighbor. The program has 5 phases: `start`, `assign_seats`, `make_path`, `check_done` and `print_results`. Table 3.2 shows the phase transitions.

We now introduce some notation to support phases. Let a rule of the form

$$phase_0 \rightarrow phase_1 :: r \ @ \ H^k \setminus H^r \iff g \mid B$$

mean that rule r can only fire if the current phase is $phase_0$; after it fires, the current phase changes to $phase_1$. A default phase transition (i.e., when no rule applies in the current phase) is represented similarly by a declaration of the form

$$phase_0 \rightarrow phase_1$$

The pseudo-code of Listing 3.10 implements the Manners program.

```
start → assign_seats :: assign_first_seat @ guest(Name,_,_) ==>
        seating(1,Name,1,0,yes), on_path(1,1,Name), next_id(2).

assign_seats → make_path :: find_seating @
    seating(Seat,Name,Id,PId), guest(Name,Sex1,Hobby),
    guest(Guest,Sex2,Hobby) \ next_id(NId) <=>
        Sex1 \= Sex2, not_chosen(Id,Guest), not_on_path(Id,Guest) |
        chosen(Id,Guest), seating(Seat+1,Guest,NId,Id),
        on_path(NId,Seat+1,Guest), next_id(NId+1).

extend_path → extend_path :: extend_path @
    seating(_,_,Id,PId), on_path(PId,Seat,Name) ==>
        not_on_path(Id,Name) | on_path(Id,Seat,Name).

extend_path → check_done.

check_done → print_results :: are_we_done @
    last_seat(LastSeat), seating(LastSeat,_,_,_) ==> true.

check_done → assign_seats.

print_results → print_results :: print_results @
    on_path(Id,LastSeat,_), last_seat(LastSeat) \
    on_path(Id,Seat,Name) <=> write(Seat:Name), nl.

print_results → print_results :: print_last_result @
    on_path(_,LastSeat,LastName), last_seat(LastSeat) <=>
        write(Seat:Name),nl.
```

Listing 3.10: Pseudo-code implementation of Miss Manners in CHR with phases

The code works as follows. The initial goal consists of `guest`/3 constraints representing all guests to be seated with their hobbies, and a `last_seat`/1 constraint with as argument the number of guests to be seated. If a single guest has multiple hobbies, there is one `guest`/3 constraint for each of these hobbies. We start in phase `start`. The program now proceeds by creating a search tree of possible seating assignments. Each node in the search tree consists of a guest being assigned a seat, and is represented by a `seating`/4 constraint. The arguments of such a `seating`/4 are respectively the seat assigned, the guest assigned to this seat, an identifier of the search tree node, and an identifier of its parent in the search tree. The `on_path`/3 and `chosen`/2 constraints make sure the same guest is not assigned a seat twice and all guests that have not been assigned a seat yet are considered only once at each choice point.

We need two auxiliary constraints `not_on_path`/2 and `not_chosen`/2 to implement negation as absence. Under the refined operational semantics of CHR, these constraints can be implemented as follows.

```
path(Id,_,Name) \ not_path(Id,Name) <=> fail.
not_path(_,_) <=> true.

chosen(Id,Name) \ not_chosen(Id,Name) <=> fail.
not_chosen(_,_) <=> true.
```

□

It is possible to implement rule priorities using phases. In a way, an example of this is the implementation of the Sudoku solver of Listing 3.8. For static priorities, the transformation is fairly straightforward. Every rule belongs to a phase corresponding to its priority. After a rule fires, the phase changes to the highest priority phase. When no rule is applicable in the current phase, it is changed to the next (lower priority) phase. Things become more complicated when dynamic rule priorities are involved. In this case, a rule belongs to a dynamic phase based on the arguments of the constraints involved in its instances. It remains unclear how the phase changes should be modeled as in theory there are infinitely many phases. In any case, while it seems possible to model rule priorities using phases, it will require a highly specialized implementation to be efficient.

Implementing phases using priorities proves to be difficult. One could consider the phase a rule belongs to as an extra constraint that needs to in the store in order for the rule to be applicable. However, to make this approach work, we need to exclude this special purpose constraint from consideration in the propagation history. For example, consider the `extend_path` rule from the Manners program. We could implement this rule in CHR^{rp} as follows:

```
1 :: extend_path @ phase(extend_path),
    seating(_,_,Id,Pid), on_path(Pid,Seat,Name) ==>
```

```
        not_on_path(Id,Name) | on_path(Id,Seat,Name).
```

The default phase transitions can be dealt with by rules of priority 2. For example:

```
2 :: phase(check_done) <=> phase(assign_seats).
```

The problem with the above approach is that when we return to a phase we were previously in, this phase is represented by a fresh `phase`/1 constraint, and so all rule instances that fired before, may fire again as the propagation history entries differ. Moreover, even if we can prevent rules from firing again, we may still *try* the rules, which is costly in general. For instance, in the above code, if we return to phase `extend_path`, we should only consider rule instances involving the most recently asserted `seating`/4 constraint. In this case, the `not_on_path`/2 guard will prevent other rule instances from firing, but they will still be tried.

We conclude that rule priorities and execution in phases are alternative, and largely incompatible, ways of expressing execution control in CHR. We consider rule priorities the more useful alternative for implementing constraint solvers, which remains the main application area of CHR.

3.6 Basic Compilation of CHR$^{\mathrm{rp}}$

This section gives an overview of the basic compilation schema for CHR$^{\mathrm{rp}}$ programs. First, in Section 3.6.1, we present a refinement of the ω_p semantics that follows the actual implementation more closely. This refinement, called the refined priority semantics and denoted by ω_{rp}, is based on the refined operational semantics ω_r of (regular) CHR and is thus also based on lazy matching and the concept of active constraints. The ω_{rp} semantics requires that each active constraint determines the actual (ground) priorities of all rules in which it may participate. In Section 3.6.2, we show how dynamic priority rules can be transformed so that this property holds for all active constraints. Finally, Section 3.6.3 gives an abstract version of the code generated for each of the ω_{rp} transitions.

3.6.1 The Refined Priority Semantics ω_{rp}

The refined priority semantics ω_{rp} is given as a state transition system. Its states are represented by tuples of the form $\langle A, Q, S, B, T\rangle_n$, where S, B, T and n are as in the ω_p semantics, A is a sequence of constraints, called the activation stack, and Q is a priority queue. In the ω_{rp} semantics, constraints are scheduled for activation at a given priority. By $c\#i : j \ @ \ p$ we denote the identified constraint $c\#i$ being tried at its j^{th} occurrence of fixed priority p. In what follows, the priority queue is considered a set supporting the operation find_min which returns one of its highest priority elements.

1. Solve $\langle [c\|A], Q, S_0 \cup S_1, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle A, Q', S_0 \cup S_1, c \wedge B, T\rangle_n$ where c is a built-in constraint, $\mathsf{vars}(S_0) \subseteq \mathsf{fixed}(B)$ is the set of variables fixed by B, and $Q' = Q \cup \{c\#i \ @ \ p \mid c\#i \in S_1 \wedge c$ has an occurrence in a priority p rule$\}$. This reschedules constraints whose matches might be affected by c.
2. Schedule $\langle [c\|A], Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle A, Q', \{c\#n\} \cup S, B, T\rangle_{n+1}$ with c a CHR constraint and $Q' = Q \cup \{c\#n \ @ \ p \mid c$ has an occurrence in a priority p rule$\}$.
3. Activate $\langle A, Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle [c\#i : 1 \ @ \ p\|A], Q \setminus \{c\#i \ @ \ p\}, S, B, T\rangle_n$ where $c\#i \ @ \ p = \mathsf{find\_min}(Q)$, and $A = [c'\#i' : j' \ @ \ p'\|A']$ with $p < p'$ or $A = \epsilon$.
4. Drop $\langle [c\#i : j \ @ \ p\|A], Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle A, Q, S, B, T\rangle_n$ if there is no j^{th} priority p occurrence of c in P.
5. Simplify $\langle [c\#i : j \ @ \ p\|A], Q, \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle C +\!\!+ A, Q, H_1 \cup S, \theta \wedge B, T\rangle_n$ where the j^{th} priority p occurrence of c is d_j in rule $$p' :: r \ @ \ H_1' \backslash H_2', d_j, H_3' \iff g \mid C$$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $p = \theta(p')$, $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathsf{chr}(H_3) = \theta(H_3')$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$. This transition only applies if the **Activate** transition does not.
6. Propagate $\langle [c\#i : j \ @ \ p\|A], Q, \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle C +\!\!+ [c\#i : j \ @ \ p \mid A], Q, \{c\#i\} \cup H_1 \cup H_2 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where the j^{th} priority p occurrence of c is d_j in $$p' :: r \ @ \ H_1', d_j, H_2' \backslash H_3' \iff g \mid C$$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $p = \theta(p')$, $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $\mathsf{chr}(H_3) = \theta(H_3')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and $t = \langle \mathsf{id}(H_1) +\!\!+ [i] +\!\!+ \mathsf{id}(H_2), \mathsf{id}(H_3), r\rangle \notin T$. This transition only applies if the **Activate** transition does not.
7. Default $\langle [c\#i : j \ @ \ p\|A], Q, S, B, T\rangle_n \overset{\omega_{rp}}{\mapsto}_P \langle [c\#i : j+1 \ @ \ p\|A], Q, S, B, T\rangle_n$ if the current state cannot fire any other transition.

Table 3.3: Transitions of ω_{rp}

The transitions of the ω_{rp} semantics are shown in Table 3.3. The main differences compared to the ω_r semantics are the following. Instead of adding new or reactivated constraints to the activation stack, the **Solve** and **Schedule**[8] transitions schedule them for activation, once for each priority at which they have occurrences. The **Activate** transition activates the highest priority scheduled constraint if it has a higher priority than the current active constraint (if any). This transition only applies if the **Solve** and **Schedule** transitions are not applicable, i.e., after processing the initial goal or a rule body. Its function is similar to that of the ω_r **Reactivate** transition, except that it also applies to constraints that have never been activated before. Noteworthy is that once a constraint is active at a given priority, it remains so at least until a rule fires or it is made passive by the **Drop** transition. Hence we should only check the priority queue for a higher priority scheduled constraint at these program points. Again, the transitions are exhaustively applied starting from an initial state $\langle G, \emptyset, \emptyset, \mathtt{true}, \emptyset\rangle_1$ with G the goal, given as a sequence.

Example 3.15 The ω_{rp} state corresponding to the ω_t (ω_p) state after the 3 **Introduce** transitions in Example 2.2 is as follows, where we write $c\#i \mathbin{@} \{p_1, \ldots, p_n\}$ as a shorthand for $\{c\#i \mathbin{@} p_1, \ldots, c\#i \mathbin{@} p_n\}$:

$$\langle \epsilon, \{\mathtt{leq(A,B)}\#1@\{1,2\}, \mathtt{leq(B,C)}\#2@\{1,2\}, \mathtt{leq(B,A)}\#3@\{1,2\}\}, \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2, \mathtt{leq(B,A)}\#3\}, \mathtt{true}, \emptyset\rangle_4$$

If $\mathtt{leq(B,C)}\#2@1$ is activated first then it finds no matching partners and is eventually dropped. If $\mathtt{leq(A,B)}\#1@1$ is activated next, then we have

$$\langle [\mathtt{leq(A,B)}\#1:1@1], \{\mathtt{leq(A,B)}\#1@2, \mathtt{leq(B,C)}\#2@2, \mathtt{leq(B,A)}\#3@\{1,2\}\}, \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2, \mathtt{leq(B,A)}\#3\}, \mathtt{true}, \emptyset\rangle_4 \overset{\omega_{rp}}{\longmapsto}_{\mathtt{leq}} (\textbf{Default})$$

$$\langle [\mathtt{leq(A,B)}\#1:2@1], \{\mathtt{leq(A,B)}\#1@2, \mathtt{leq(B,C)}\#2@2, \mathtt{leq(B,A)}\#3@\{1,2\}\}, \{\mathtt{leq(A,B)}\#1, \mathtt{leq(B,C)}\#2, \mathtt{leq(B,A)}\#3\}, \mathtt{true}, \emptyset\rangle_4 \overset{\omega_{rp}}{\longmapsto}_{\mathtt{leq}} (\textbf{Simplify})$$

$$\langle [\mathtt{A} = \mathtt{B}], \{\mathtt{leq(A,B)}\#1@2, \mathtt{leq(B,C)}\#2@2, \mathtt{leq(B,A)}\#3@\{1,2\}\}, \{\mathtt{leq(B,C)}\#2\}, \mathtt{true}, \emptyset\rangle_4 \overset{\omega_{rp}}{\longmapsto}_{\mathtt{leq}} (\textbf{Solve})$$

$$\langle \epsilon, \{\mathtt{leq(A,B)}\#1@2, \mathtt{leq(B,C)}\#2@\{1,2\}, \mathtt{leq(B,A)}\#3@\{1,2\}\}, \{\mathtt{leq(B,C)}\#2\}, \mathtt{A} = \mathtt{B}, \emptyset\rangle_4$$

This last transition reschedules the $\mathtt{leq(B,C)}\#2$ constraint at priorities 1 and 2. None of the remaining constraints in the schedule lead to a rule firing. □

3.6.2 Transforming Dynamic Priority Rules

In the description of the ω_{rp} semantics, we have assumed that every constraint knows the priorities of all rules in which it may participate. For rules with a dynamic priority, this is obviously not always the case.

[8] The **Schedule** transition corresponds to the **Activate** transition in ω_r.

Example 3.16 Consider the rule

```
X+Y :: r @ a(X,Z) \ b(Y,Z), c(X,Y) <=> d(X).
```

In this case, a $\mathtt{c}(x, y)$ constraint with ground arguments x and y knows the priority of the instances of r in which it may participate, but neither $\mathtt{a}/2$ nor $\mathtt{b}/2$ constraints do. Given an active $\mathtt{a}/2$ constraint, we need to combine (join) it with either a $\mathtt{b}/2$ or a $\mathtt{c}/2$ constraint to determine the actual priority. □

In this section, we present a pseudo-code source-to-source transformation to transform a program such that the required property is satisfied. In what follows, we refer to the *join order* for a given constraint occurrence, which is the order in which the partner constraints for this occurrence are retrieved (by nested loops). We consider a join order Θ to be a permutation of $\{1, \ldots, n\}$ where n is the number of heads of the rule. Now, consider a dynamic priority rule

$$p :: r \ @ \ C_1, \ldots, C_i \backslash C_{i+1}, \ldots, C_n \iff g \mid B$$

an active head C_j, a join order Θ with $\Theta(1) = j$ and a number k, $1 \leq k \leq n$ such that the first k heads, starting with C_j and following join order Θ, determine the rule priority. We rewrite rule r as follows (for every j, $1 \leq j \leq n$):

$$\begin{array}{ll}
1 :: r_j \ @ & C_{\Theta(1)} \# Id_1, \ldots, C_{\Theta(k)} \# Id_k \implies \\
& \quad r\text{-}\mathtt{match}_j(Id_1, \ldots, Id_k, \mathit{Vars}) \ \mathtt{pragma\ passive}(Id_2), \ldots, \mathtt{passive}(Id_k) \\
1 :: r'_j \ @ & r\text{-}\mathtt{match}_j(Id_1, \ldots, Id_k, \mathit{Vars}) \iff \\
& \qquad\qquad \mathtt{ground}(p) \mid r\text{-}\mathtt{match}'_j(Id_1, \ldots, Id_k, \mathit{Vars}) \\
p :: r''_j \ @ & r\text{-}\mathtt{match}'_j(Id_1, \ldots, Id_k, \mathit{Vars}), C_{\Theta(k+1)} \# Id_{k+1}, \ldots, C_{\Theta(n)} \# Id_n \implies \\
& \quad \mathtt{alive}(Id_1), \ldots, \mathtt{alive}(Id_k), g \mid \mathtt{kill}(Id_{\Theta^{-1}(i+1)}), \ldots, \mathtt{kill}(Id_{\Theta^{-1}(n)}), B \\
& \quad \mathtt{pragma\ passive}(Id_{k+1}), \ldots, \mathtt{passive}(Id_n), \\
& \qquad\qquad \mathtt{history}([Id_{\Theta^{-1}(1)}, \ldots, Id_{\Theta^{-1}(n)}, r])
\end{array}$$

where *Vars* are the variables shared by the first k heads on the one hand, and the remaining heads, the guard, the body and the priority expression on the other, i.e., $\mathit{Vars} = \big(\cup_{i=1}^{k} \mathsf{vars}(C_{\Theta(i)})\big) \cap \big(\big(\cup_{i=k+1}^{n} \mathsf{vars}(C_{\Theta(i)})\big) \cup \mathsf{vars}(g \wedge B \wedge p)\big)$. The first rule generates a partial match that knows its priority once the necessary arguments are ground (fixed). It runs at the highest possible value of the dynamic priority expression.[9] The second rule ensures that the priority expression is ground before the partial match is scheduled at its dynamic priority. The rule runs at the same priority as the first one. Finally, the third rule extends the partial match (with ground priority) into a full match. There we check whether all constraints in the partial match are still alive (calls to `alive`/1), and delete the removed heads (calls to `kill`/1). The *pragma*[10] `passive`/1 denotes that a given head is passive, i.e.,

[9] We assume 1 is an upper-bound. A tighter one can be used instead if such is known.
[10] Most CHR systems support compiler directives by using the keyword `pragma`.

no occurrence code is generated for it (see further in Section 3.6.3). The pragma `history`/1 states the tuple layout for the propagation history. All rule copies share the same history which ensures that each instance of the original rule can fire only once.

Example 3.17 Given the rule `r` of Example 3.16 and join orders $\Theta_1 = [1,2,3]$, $\Theta_2 = [2,3,1]$ and $\Theta_3 = [3,2,1]$.[11] Furthermore assuming we schedule at a dynamic priority as soon as we know it, we generate the rules of Listing 3.11.

```
  1 :: r1 @ a(X,Z) #Id1, b(Y,Z) #Id2 ==>
       r-match1(Id1,Id2,X,Y) pragma passive(Id2).
  1 :: r1' @ r-match1(Id1,Id2,X,Y) <=>
       ground(X+Y) | r-match1'(Id1,Id2,X,Y).
X+Y :: r1'' @ r-match1'(Id1,Id2,X,Y), c(X,Y) #Id3 ==>
       alive(Id1), alive(Id2) | kill(Id2), kill(Id3), d(X)
       pragma passive(Id3), history([Id1,Id2,Id3],r).

  1 :: r2 @ b(Y,Z) #Id1, c(X,Y) #Id2 ==>
       r-match2(Id1,Id2,X,Y,Z) pragma passive(Id2).
  1 :: r2' @ r-match2(Id1,Id2,X,Y,Z) <=>
       ground(X+Y) | r-match2'(Id1,Id2,X,Y,Z).
X+Y :: r2'' @ r-match2'(Id1,Id2,X,Y,Z), a(X,Z) #Id3 ==>
       alive(Id1), alive(Id2) | kill(Id1), kill(Id2), d(X)
       pragma passive(Id3), history([Id3,Id1,Id2,r]).

  1 :: r3 @ c(X,Y) #Id1 ==> r-match3(Id1,X,Y).
  1 :: r3' @ r-match3(Id1,X,Y) <=> ground(X+Y) | r-match3'(Id1,X,Y).
X+Y :: r3'' @ r-match3'(Id1,X,Y), b(Y,Z) #Id2, a(X,Z) #Id3 ==>
       alive(Id1) | kill(Id1), kill(Id2), d(X)
       pragma passive(Id2), passive(Id3), history([Id3,Id2,Id1,r]).
```

Listing 3.11: Example output of the dynamic priority rule transformation

Note that since `r` is a simpagation rule, a propagation history is not necessary. We only show it for illustrative purposes. □

The proposed translation schema implements a form of eager matching in that all r-`match`$_j$ constraints are generated eagerly at the highest priority before one is fired. This approach resembles the TREAT matching algorithm (Miranker 1987). Also similar to the TREAT algorithm and unlike the RETE algorithm (Forgy 1982), we allow different join orders for each active head.

[11] By slight abuse of syntax, we denote $\Theta(1) = \theta_1, \ldots, \Theta(n) = \theta_n$ by $\Theta = [\theta_1, \ldots, \theta_n]$.

3.6.3 Compilation Schema

Now that we have shown how a program can be transformed such that each constraint knows the priorities of all rules in which it may participate, we are ready to present the compilation schema. The generated code follows the ω_{rp} semantics closely. In what follows, we assume the host language is Prolog, although the compilation process easily translates to other host languages as well. We note that the generated code presented in this section, closely resembles that of regular CHR under the refined operational semantics, as described in for example (Schrijvers 2005). The differences correspond to those between ω_r and ω_{rp} as given in Section 3.6.1. An example of the compiled code can be found in Appendix A.

CHR Constraints

Whenever a new CHR constraint is asserted, it is scheduled at all priorities at which it may fire (**Schedule** transition). Furthermore, it is attached to its variables for the purpose of facilitating the **Solve** transition. In Prolog this is done using attributed variables. The idea is similar to that of subscribing to event notifiers. Finally, the constraint is inserted into all indexes on its arguments. Schematically, the generated code is as shown in Listing 3.12.

```
c(X1,...,Xn) :- GenerateSuspension, S = Suspension,
                schedule(p1,c/n_prio_p1_occ_1_1(S)),
                ...
                schedule(pm,c/n_prio_pm_occ_1_1(S)),
                AttachToVariables, InsertIntoIndexes.
```

Listing 3.12: Generated code for a new CHR constraint

The *GenerateSuspension* code creates a data structure (called the *suspension term* in CHR terminology) for representing the constraint in the constraint store. It has amongst others fields for the constraint identifier, its state (dead or alive), its propagation history,[12] its arguments, and pointers for index management. The scheduling code consists of insertions of calls to the code for the first occurrence of each priority p_i $(1 \leq i \leq m)$ into the priority queue. With respect to the usual code for CHR constraints under the ω_r semantics, we have added the `schedule/2` calls and removed the call to the code for the first occurrence of the constraint.

Built-in Constraints

Built-in constraints are dealt with by the underlying constraint solver, in this case the Prolog Herbrand solver. Whenever this solver binds a variable to another

[12] We use a distributed propagation history, as in the K.U.Leuven CHR system (Schrijvers 2005).

variable or a term (during unification), a so-called *unification hook* is called (see (Holzbaur 1992; Demoen 2002)). In this hook, the CHR part of the **Solve** transition is implemented. It consists of reattaching the affected constraints, updating the indexes, and scheduling the affected constraints again at each priority for which they have occurrences.

Occurrence Code

For each constraint occurrence, a separate predicate is generated, implementing the **Simplify** and **Propagate** transitions. Its clauses are shown schematically in Listing 3.13. The approach is very similar to how occurrences are compiled under the refined operational semantics of CHR. The differences are that only the occurrences of the same priority are linked, where occurrences with a dynamic priority are assumed to run at different priorities, and the priority queue is checked (`check_activation`/1) after each rule firing. The code shown is for the j^{th} priority p occurrence of the c/n constraint which is in an m-headed rule. The indices $r(1), \ldots, r(i)$ refer to the removed heads.

The *HeadMatch* call checks whether the newly looked-up constraint matches with the corresponding rule head and with the already matched head constraints. A list of all candidate constraints for the next head is returned by *LookupNext*/1. *RemainingGuard* is the part of the guard that has not already been tested by the *HeadMatch* calls. Propagation history checking and extending is handled by respectively *HistoryCheck* and *AddToHistory*. After having gone through all rule instances for the given occurrence, the next occurrence is tried (**Default**) or the activation call returns (**Drop**). The `check_activation`/1 call in the occurrence code checks whether a constraint occurrence is scheduled at a higher priority than the current one. It implements the **Activate** transition.

3.7 Optimizing the Compilation of CHR$^{\text{rp}}$

We now present the main optimizations implemented in the CHR$^{\text{rp}}$ compiler. The proposed optimizations mainly improve constant factors, but might cause complexity improvements for some programs as well. We start with optimizations that reduce the number of priority queue operations. We note that such operations may have a higher than constant cost. The optimizations are illustrated on an example program in Appendix A.

3.7.1 Reducing Priority Queue Operations

A first optimization consists of only scheduling the highest priority occurrence of every new constraint. Only when the constraint has been activated at this priority and has gone through all of its occurrences without being deleted, it is scheduled

```
c/n_prio_p_occ_j_1(S1) :-
        (   alive(S1), HeadMatch, LookupNext(S̄2)
        ->  c/n_prio_p_occ_j_2(S̄2,S1).
        ;   c/n_prio_p_occ_j + 1_1(S1)
        ).

c/n_prio_p_occ_j_2([S2|S̄2],S1) :-
        (   alive(S2), S2 \= S1, HeadMatch, LookupNext(S̄3)
        ->  c/n_prio_p_occ_j_3(S̄3,S2,S̄2,S1)
        ;   c/n_prio_p_occ_j_2(S̄2,S1)
        ).
c/n_prio_p_occ_j_2([],S1) :- c/n_prio_p_occ_j + 1_1(S1).

...

c/n_prio_p_occ_j_m([Sm|S̄m],Sm−1,...,S1) :-
        (   alive(Sm), Sm \= S1, ..., Sm \= Sm−1,
            HeadMatch, RemainingGuard, HistoryCheck
        ->  AddToHistory, kill(Sr(1)), ..., kill(Sr(i)),
            Body, check_activation(p),
            (   alive(S1)
            ->  (   ...
                ... (   alive(Sm−1)
                    ->  c/n_prio_p_occ_j_m(S̄m,...,S1)
                    ;   c/n_prio_p_occ_j_m − 1(S̄m−1,...,S1)
                    )
                ...
            ;   true
            )
        ;   c/n_prio_p_occ_j_m(S̄m,Sm−1,...,S1)
        ).
c/n_prio_p_occ_j_m([],_,S̄m−1,...,S1) :-
        c/n_prio_p_occ_j_m − 1(S̄m−1,...,S1).
```

Listing 3.13: Generated occurrence code

for the next priority. This is a simple extension of how we linked occurrences of equal priority in the occurrence code.

In the basic compilation scheme, it is checked whether a higher priority scheduled constraint exists after each rule firing. In a number of cases, this is not needed. If the active constraint is removed, it is popped from the top of the activation stack and the activation check that caused it to be activated in the first place, checks again to see if other constraints are ready for activation. So, since a priority queue check will take place anyway, there is no need to do this twice. If the body of a rule does not contain CHR constraints with a priority higher than the current one, nor built-in constraints that can trigger any CHR constraints to be scheduled at a higher priority, then after processing the rule body, the active constraint remains active and we do not need to check the priority queue. We denote the above optimizations by *reduced activation checking*.

Building further on this idea, we note that by analyzing the body, we can sometimes determine which constraint will be activated next. Instead of scheduling it first and then checking the priority queue, we can activate it directly at its highest priority. We call this *inline activation*. Inline activation is not limited to one constraint: we can directly activate all constraints that have the same highest priority. Indeed, when the first of these constraints returns from activation, the priority queue cannot contain any constraint scheduled at a higher priority, because such a constraint would have been activated before returning.

Example 3.18 We illustrate the applicability of the proposed optimizations on the `leq` program given in Listing 3.1. The `leq`/2 constraint has 5 occurrences at priority 1 and 2 at priority 2. New `leq`/2 constraints are only scheduled at priority 1. Only if an activated constraint has passed the 5^{th} priority 1 occurrence, it is scheduled at priority 2. For the first three priority 1 occurrences, as well as for the removed occurrence in the `idempotence` rule, the active constraint is removed and so there is no need to check the priority queue after firing the rule body. Since the body of the remaining priority 1 occurrence equals `true`, no higher priority constraint is scheduled and so we do not need to check the queue here either. Finally, for the `transitivity` rule we have that the only constraint in the body has a higher priority occurrence than the current active occurrence, and so we can apply inline activation there. □

3.7.2 Late Indexing

Similar to an optimization from regular CHR, we can often postpone storage of constraints, reducing cost if the constraint is removed before the storage operations are to be applied. We extend the late storage concept of (Holzbaur et al. 2005) to late indexing, where we split up the task of storing a constraint into the subtasks of inserting it into different indexes. The main idea is that an active constraint can only be suspended by another constraint for occurrences of that constraint in

rules of a higher priority. This implies that when a constraint is active at a given current priority, it should only be stored in those indexes that are used by higher priority rules.

Example 3.19 In the `leq` program (Listing 3.1), the `leq`/2 constraints are indexed

- on the combination of both arguments (`antisymmetry` and `idempotence`);
- on the first argument and on the second argument (`transitivity`);
- on the constraint symbol for the purpose of showing the constraint store.

By using late indexing, new `leq`/2 constraints are not indexed at the moment they are asserted, but only scheduled (and this only at priority 1). When an active `leq`/2 constraint 'survives' the 5th priority 1 occurrence, it is indexed on the combination of both arguments and rescheduled at priority 2. We can postpone the indexing this long because only one constraint can be on the execution stack for each priority and hence all partner constraints have either been indexed already, or still need to be activated. Only after a reactivated `leq`/2 constraint has passed the second priority 2 occurrence, it is stored in the remaining indexes. Note that our approach potentially changes the execution order of the program, which can sometimes contribute to changes in the running time (in either direction). □

3.7.3 Passive Occurrences

In this section, we show that some constraint occurrences can be made passive, which allows us to avoid the overhead of looking up partner constraints, and sometimes also the overhead related to scheduling and indexing. We first give an example and then present the general approach.

Example 3.20 (Naive Union-Find) Listing 3.14 shows a naive CHR^{rp} implementation of the union-find algorithm (see e.g. (Tarjan and van Leeuwen 1984)) and is adapted from (Schrijvers and Frühwirth 2006).[13]

```
1 :: findNode @ X ~> PX \ find(X,R) <=> find(PX,R).
2 :: findRoot @ find(X,R) <=> R = X.
3 :: linkEq   @ link(X,X) <=> true.
4 :: link     @ link(X,Y) <=> Y ~> X.
5 :: union    @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).
```

Listing 3.14: Naive union-find in CHR^{rp}

[13] Because of the rule priorities, we do not need the `root`/1 constraints, as is the case for CHR under the ω_r semantics.

The input to this algorithm consists of `union`/2 and `find`/2 constraints, representing the corresponding operations. The ∼>/2 constraint is a *data* constraint and is used as internal representation for linked items. The `link`/2 constraint is an *operation* constraint that causes its arguments to be linked.[14]

By looking at the rule bodies, we see that the ∼>/2 constraint is only asserted at priority 4 whereas its partner constraint in rule `findNode` (`find`/2) is unconditionally removed after priority 2 by rule `findRoot`. Therefore, whenever an ∼>/2 constraint is asserted, it will not be able to fire rule `findNode` and its occurrence in that rule can be made passive. Hence we do not need to schedule the constraint once it is asserted, but we do need to store it. Note that the ∼>/2 constraints could also appear in the initial goal. We can however consider the goal as the body of a rule that runs at the lowest possible priority. □

We now give the general approach. Consider a constraint occurrence $c : j \ @ \ p$ in some rule r and let p_{max} be the highest priority at which a c constraint can be asserted by any rule. For constraints that only appear in the goal, $p_{\mathsf{max}} = +\infty$. For constraints with non-ground indexed arguments, p_{max} equals the highest priority at which either a c constraint, or a built-in constraint is asserted. Let p_{rm} be the highest priority at which one of the partner constraints of $c : j \ @ \ p$ is unconditionally removed. If no such priority exists, $p_{\mathsf{rm}} = +\infty$. We assume that $p < p_{\mathsf{rm}}$, otherwise rule r can never fire. If $p_{\mathsf{rm}} < p_{\mathsf{max}}$ then $c : j \ @ \ p$ can be made passive and must be stored for this occurrence before any rule is tried at priority p.

The correctness of this approach is shown as follows. Consider a rule instance $\theta(r)$ that is applicable, but is missed because we made $c : j \ @ \ p$ passive. Let c' be the most recently asserted (or reactivated) constraint in $\theta(r)$. If $c' = c$ then all partner constraints of c must have been asserted before c and have not been removed thereafter. Clearly, this contradicts the assumption that $p_{\mathsf{rm}} < p_{\mathsf{max}}$. If $c' \neq c$ then because c is stored at the relevant indexes, the rule instance is found by c'. This reasoning easily extends to multiple passive heads.

3.8 Benchmark Evaluation

In this section, we evaluate the performance of our system on some benchmarks. The evaluation shows the merits of our optimizations, as well as the competitiveness of our system with respect to the state-of-the-art K.U.Leuven CHR system (Schrijvers and Demoen 2004), which is based on the refined operational semantics of (regular) CHR.

[14] When using CHR as a general purpose programming language, one often uses two types of constraints: (passive) *data* constraints and (active) *operation* constraints.

Less-or-Equal The `leq` benchmark uses the program of Listing 3.1 and for given n, the initial goal $G = G_1 \cup G_2$ with

$$G_1 = \{\texttt{leq}(\texttt{X}_1, \texttt{X}_2), \ldots, \texttt{leq}(\texttt{X}_{n-1}, \texttt{X}_n)\} \wedge G_2 = \{\texttt{leq}(\texttt{X}_n, \texttt{X}_1)\}$$

From the goal G, a final state is derived in which $\texttt{X}_1 = \texttt{X}_2 = \ldots = \texttt{X}_{n-1} = \texttt{X}_n$.

Because of the *batch* semantics of CHR$^{\text{rp}}$ (i.e., the constraints from the goal are all inserted into the store before the first of them is activated), we achieve an almost linear time complexity (in n) for this benchmark because of the order in which constraints are activated.[15] Noteworthy is that by using the late indexing optimization, we get a higher complexity because the necessary partner constraints for the optimal firing order are not yet stored. This is illustrated in Figure 3.1, which shows runtimes for the `leq` benchmarks in three different setups. In particular, we have tested our CHR$^{\text{rp}}$ system with late indexing (Priority, +LI) and without late indexing (Priority, -LI), and have also compared with regular CHR under the ω_r semantics (Refined) using the K.U.Leuven CHR system (with all optimizations turned on). The benchmarks were run on a Pentium IV, 2.8GHz running SWI-Prolog version 5.6.55. The results do not include garbage collection times. Even with late indexing, our implementation performs better than the K.U.Leuven CHR system for large enough values of n. This is mainly due to the non-ground hashing that is used for indexing in our system (see further in Section 6.2.2).

Optimizations Table 3.4 shows benchmark results for various programs where the effect on the runtime of each of the optimizations is measured. The runtimes are given as percentages of the runtime of the unoptimized version for each program. For the unoptimized and fully optimized versions, we also give times in seconds. We use the same setup as in the previous paragraph. The `loop` benchmark consists of the following two rules (and does not rely on priorities):

```
1 :: a(X) <=> X > 0 | a(X-1).        1 :: a(0) <=> true.
```

and initial goal $\{\texttt{a}(2^{20})\}$. The `leq` benchmark is based on the one presented in the previous paragraph. However, to measure the effects of late indexing properly, we ensure that the same rule instances fire in both the versions with and without late indexing. We do so by first asserting the subgoal G_1, waiting for a fixpoint, and only then asserting subgoal G_2. We have measured for $n = 80$. The `dijkstra` benchmark uses the program of Example 3.2 with a graph of 2^{15} nodes and $3 \cdot 2^{15}$ edges; the `union-find` benchmark uses the program of Example 3.20 with 2^{12} random `union`/2 constraints over an equal number of elements; and finally the `sudoku` benchmark uses the program of Example 3.6 and solves a puzzle in which initially 16 cells have a value (Figure 3.2).[16]

[15] More precisely, constraints are activated in LIFO order.

[16] This puzzle can be solved without backtracking, but this requires a stronger form of consistency, and relies on the observation that a symmetric solution can be found by switching numbers

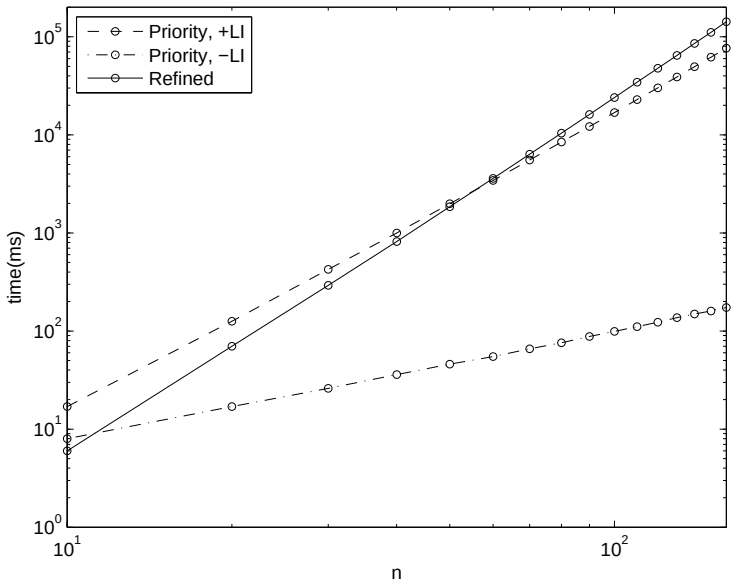


Figure 3.1: Benchmark results for `leq`

The benchmarks are executed with the late indexing (LI), inline activation (IA) and reduced activation checking (RAC) optimizations switched on and off.

The inline activation analysis assumes that dynamic priority rules run at the highest possible value of the priority expression. It currently assumes this value is 1, but a bounds analysis or a user declaration can give a tighter upper-bound. In the `dijkstra` and `sudoku` benchmarks, we have used a tight upper-bound of 2 for the dynamic priority rules. The passive analysis applied to the `union-find` benchmark cuts off another 1% and reduces the runtime with full optimization to about 16% of the runtime without optimization. In the `loop` benchmark, inline activation only has a strong effect in combination with reduced activation checking: the combined optimizations reduce the runtime by 42% whereas the individual optimizations only cause a reduction of respectively 11% and 2%. The late indexing optimization can change the execution order. We have already shown how this affects the `leq` benchmark. Similarly, it also affects the `sudoku` benchmark which has (amongst others) 11% more rule firings in the version *with* late indexing, hence the increase in runtime. Moreover, late indexing only reduces the amount of index insertions by 3% in this benchmark. Therefore, in this case we get the best result,

8 and 9.

1								5
				3				
		2		4				
	3	4				7		
			2		6			1
2					5			
	7						3	
					1			

Figure 3.2: Benchmark Sudoku puzzle

LI	IA	RAC	`loop`	`leq`	`dijkstra`	`union-find`	`sudoku`
			28.82s	14.10s	45.93s	18.79s	16.17s
		√	89%	98%	98%	93%	98%
	√		98%	93%	98%	82%	99%
√			46%	65%	95%	39%	114%
√	√	√	8%	57%	91%	17%	111%
√	√	√	2.42s	8.04s	41.84s	3.24s	17.90s

Table 3.4: Benchmark results

namely a runtime of 15.68 seconds, when all optimizations except for late indexing are turned on.

We also compare CHR$^{\text{rp}}$ against the K.U.Leuven CHR system under the ω_r semantics. For `leq`, `loop` and `union-find`, we execute the same code ignoring priorities (though sometimes relying on rule order). For `dijkstra` and `sudoku` the K.U.Leuven CHR code encodes the behavior obtained using priorities in CHR$^{\text{rp}}$ by other methods. Hence the rules are more involved. The `leq` benchmark takes about 23% less using CHR$^{\text{rp}}$ and the `union-find` benchmark takes 62% more time. The `loop` benchmark takes about 6.4 times longer in our system compared to the code generated by the K.U.Leuven CHR system, which corresponds to a pure Prolog loop. The main remaining overhead is the generation and destruction of internal data structures (suspension terms), which is avoided in K.U.Leuven CHR. Comparison for the `sudoku` benchmark is difficult because the search trees are different. In this particular case, K.U.Leuven CHR is about 28% faster than our CHR$^{\text{rp}}$ system (without late indexing), but also fires 14% fewer rules.

For the `dijkstra` benchmark, we compared with the CHR program given in (Sneyers et al. 2006a).[17] Our implementation runs about 2.4 times slower than the (regular) CHR implementation, but it is also arguably more high-level. Note-

[17]For a fair comparison, we use a combination of Fibonacci heaps for the dynamic priorities, and an array for static priorities 1 and 2, as priority queue.

worthy is the following optimization, implemented in (Sneyers et al. 2006a) and reformulated here in terms of our CHR^{rp} implementation. The rule

```
1 :: dist(V,D1) \ dist(V,D2) <=> D1 =< D2 | true.
```

removes the $\texttt{dist}(\texttt{V}, \texttt{D}_2)$ constraint which might still be scheduled at priority $\texttt{D}_2 + 2$. After firing the rule, the $\texttt{dist}(\texttt{V}, \texttt{D}_1)$ constraint is scheduled at priority $\texttt{D}_1 + 2$. Instead of first (lazily) deleting a scheduled item, and then inserting a new one, the cheaper `decrease_key` operation can be used instead (because $\texttt{D}_1 \leq \texttt{D}_2$). Compared to an altered version of the original CHR implementation in which this optimization is turned off, our code is (only) 15% slower.[18]

3.9 Related Work

Rule Priorities Rule priorities are found in many rule based languages. Production rule systems like CLIPS (Giarratano 2002), Jess (Friedman-Hill 2007) or JBoss Rules (Proctor et al. 2007) use rule priorities (*salience*) as part of conflict resolution. These priorities are either integers, or a partial order between rules as in the *active database system* Starburst (Widom 1996). Most production rule systems use the RETE matching algorithm (Forgy 1982), which is an eager matching algorithm that exhibits high memory requirements, but allows for an easy implementation of priority schemes. A lazy matching algorithm called LEAPS (Miranker et al. 1990) is used by a few production rule systems, such as Venus (Browne et al. 1994) and JBoss Rules (in an experimental stage). This algorithm is similar to what is used by CHR implementations based on the refined operational semantics. It seems that in these systems, priorities are only used relative to the active constraint (*dominant object* in LEAPS terminology), thus not allowing 'global' priorities like the ones proposed in this work.

Priorities have also been introduced in term rewriting systems (*Priority Rewrite Systems* (Baeten et al. 1987)). There, rule priorities are used to resolve conflicts that lead to non-confluent behavior. More recently, this idea has also been applied to term-graphs (Caferra et al. 2006). Brewka and Eiter (1999) use rule priorities to choose between alternative answer sets in answer set programming and García and Simari (2004) use priorities for a similar purpose in the context of defeasible logic programming.

A bottom-up logic programming language with prioritized rules is presented by Ganzinger and McAllester (2002). The language supports dynamic priorities that depend on the first head in the rule. It only computes those (partial) matches that have the current highest priority, but stores them in a RETE-like fashion.

[18] More precisely, we have replaced the `decrease_key` operation by an insertion and have ensured that only one labeling step is done for each node.

Priorities in C(L)P Systems Many Constraint (Logic) Programming systems offer some form of priorities (see (Schulte and Stuckey 2004)). The SICStus finite domain solver `clp(fd)` (Carlsson et al. 1997) uses two priority levels: the highest priority is reserved for constraint propagators implemented by *indexicals*. Specialized algorithms implementing global constraints are scheduled at the lowest priority.

ILOG CPLEX (ILOG 2001a) uses priorities to select variables for branching during branch and bound optimization. ILOG Solver (ILOG 2001b) appears to be using only one *propagation queue* although its manual suggests that constraints can be pushed onto a *constraint priority queue* at a given priority.

The ECL^{i}PS^e Constraint Logic Programming system (Wallace et al. 1997) supports execution at 12 different priorities. A goal G is executed at a given priority p by using `call_priority`(G, p). The priority system is shared by all constraint libraries, which allows for a form of global control over cooperating constraint solvers. The Extended Constraint Handling Rules library (`ech`) of ECL^{i}PS^e uses the priority system to support a form of static constraint priorities. See Section 3.5.2 for more details.

3.10 Conclusion

We have extended the Constraint Handling Rules language with user-defined rule priorities. The extended language, called CHR$^{\mathrm{rp}}$, supports a high-level, flexible and declarative form of execution control. It allows the programmer to write programs that are more concise, but perhaps not confluent under the theoretical operational semantics ω_t of CHR, while still offering a high-level and declarative reading. The latter is in contrast with the low-level, procedural nature of execution under the refined operational semantics ω_r. In CHR$^{\mathrm{rp}}$, all execution control information is in the priority annotations, which creates a clear separation of the logic and control aspects of a CHR$^{\mathrm{rp}}$ program.

We have formalized the syntax and semantics of CHR$^{\mathrm{rp}}$ and investigated its theoretical properties. We have shown how common CHR programming patterns translate to CHR$^{\mathrm{rp}}$ and have compared rule priorities with other forms of execution control. Next, we presented a compilation schema for CHR$^{\mathrm{rp}}$. We have shown the feasibility of implementing rules with both static and dynamic rule priorities using a lazy matching approach, in contrast with eager matching as implemented by the RETE algorithm and derivatives. We have proposed various ways to optimize the generated code. Some of the optimizations are completely new (those related to reducing priority queue operations), while others are refinements of previously known optimizations for regular CHR (namely late indexing and the passive analysis). The optimizations have been shown to be effective on benchmarks, which furthermore indicate that our implementation has a comparable performance w.r.t. the state-of-the-art K.U.Leuven CHR system, while offering

a much more high-level form of execution control.

In the next chapter, CHR^{rp} is combined with $\text{CHR}^{\vee}$ (Abdennadher and Schütz 1998) and extended with *branch priorities*, leading to a complete solution to execution control in CHR that deals with both *propagator* and *search* priorities.

Chapter 4

A Flexible Search Framework on top of CHR^{rp}

This chapter introduces a framework for the specification of tree search strategies in CHR with disjunction (CHR$^{\vee}$). We support the specification of common exploration strategies such as depth-first, breadth-first and best-first, as well as constrained optimization by means of branch & bound search. The framework is given as an extension of CHR with rule priorities (CHR^{rp}) in which each branch of the search tree is assigned a *branch priority*. This approach leads to a uniform solution to execution control in CHR.

4.1 Introduction

CHR aims at being a high-level language for implementing constraint solvers. Indeed, it is excellent at representing the propagation logic of constraint solvers: the notion of a constraint propagator corresponds with a CHR rule. The CHR$^{\vee}$ language extension (Abdennadher and Schütz 1998) also presents a high-level means to express the search aspect of constraint solving. However, due to the non-deterministic operational semantics of these features, CHR is but an abstraction of constraint solving. Most constraint problems are of too big a size to be naively entrusted to a non-deterministic solving process. Rather, solving strategies must be specified to cleverly direct the solving process and prune the search space early and eagerly. An appropriate solving strategy can indeed reduce the solving cost by many orders of magnitude and make the difference between an infeasible and a practical approach.

It is for this reason that most state-of-the-art constraint solvers offer the means to select and/or specify the desired solving strategy. For instance, the `clp(fd)` library of SICStus Prolog (Carlsson et al. 1997) contains 18 documented options,

many of which are parameterized and/or can be combined, to influence the strategy of the enumeration predicate `labeling/2`. The solving strategy often falls apart into two distinct aspects: propagator priorities for conjunctions and search priorities for disjunctions. Propagator priorities can be specified by means of rule priorities: these have been studied in Chapter 3. In this chapter, we add the missing piece: search priorities.

The main contributions of this chapter are the following:

1. We present CHR$^{brp}_{\vee}$, a high-level approach for specifying the control flow in CHR$^{\vee}$ (Section 4.3). CHR$^{brp}_{\vee}$ extends CHR$^{\vee}$ with both ***b**ranch* and ***r**ule **p**riorities*.

2. We show how to express standard tree exploration strategies such as depth-first, breadth-first, depth-first iterative deepening and limited discrepancy search in CHR$^{brp}_{\vee}$ (Section 4.4).

3. We show how conflict-directed backjumping can be realized by extending our framework with justifications (Section 4.5). Our work extends that of Wolf et al. (2007) by not restricting the exploration strategy to left-to-right depth-first, and by addressing correctness and optimality.

4.2 CHR with Disjunction

Constraint Handling Rules is extended with disjunctions in rule bodies in (Abdennadher and Schütz 1998) (see also (Abdennadher 2000) and (Abdennadher 2001, Chapter 5)). The resulting language is denoted by CHR$^{\vee}$. The syntax of CHR$^{\vee}$ is the same as that of regular CHR, except that rule bodies are formulas built from atoms by conjunctions and disjunctions.

We define a theoretical operational semantics $\omega_t^{\vee}$ for CHR$^{\vee}$, which extends the ω_t semantics of CHR. An $\omega_t^{\vee}$ execution state is a multi-set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ of ω_t execution states. Each element $\sigma_i \in \Sigma$ represents an alternative solution. Table 4.1 gives the transitions defined on $\omega_t^{\vee}$ execution states:

- The **Derive** transition applies an ω_t transition to one of the alternatives in the $\omega_t^{\vee}$ state;

- The **Split** transition splits up an alternative with a disjunction in its goal, into two or more alternatives, one for each disjunct;

- The **Drop** transition removes failed alternatives from the search tree and is introduced to support pruning of the search tree, for example during conflict-directed backjumping (see Section 4.5). This pruning respects the declarative semantics of CHR$^{\vee}$ (see (Abdennadher 2001, Chapter 5)).

An example CHR$^{\vee}$ program is given below.

1. Derive $\{\sigma\} \uplus \Sigma \overset{\omega_t^\vee}{\mapsto}_P \{\sigma'\} \uplus \Sigma$ if there exists a transition $\sigma \overset{\omega_t}{\mapsto}_P \sigma'$.
2. Split $\{\langle\{G_1 \vee \ldots \vee G_m\} \uplus G, S, B, T\rangle_n\} \uplus \Sigma \overset{\omega_t^\vee}{\mapsto}_P \{\langle G_1 \uplus G, S, B, T\rangle_n, \ldots, \langle G_m \uplus G, S, B, T\rangle_n\} \uplus \Sigma$.
3. Drop $\{\sigma\} \uplus \Sigma \overset{\omega_t^\vee}{\mapsto}_P \Sigma$ if $\sigma = \langle G, S, B, T\rangle_n$ is a failed execution state, i.e., $\mathcal{D} \models \neg\bar{\exists}_\emptyset B$.

Table 4.1: Transitions of $\omega_t^\vee$

Example 4.1 (4-queens) A solver for the 4-queens problem can be written in CHR$^\vee$ as shown in Listing 4.1. As goal we use {`queens`}.

```
queens <=> row(1), row(2), row(3), row(4).
row(R) <=> queen(R,1) ∨ queen(R,2) ∨ queen(R,3) ∨ queen(R,4).

queen(_ ,C1), queen(_ ,C2) ==> C1 =\= C2.
queen(R1,C1), queen(R2,C2) ==> abs(R1 - R2) =\= abs(C1 - C2).
```

Listing 4.1: CHR$^\vee$ implementation of 4-queens

Here, a `queen`(r, c) constraint means that there is a queen placed on the field with row r and column c. The first rule states that there are four rows. The second rule states that there is a queen in one of the columns for each row. The remaining two rules ensure that no two queens are in conflicting positions. The first of them makes sure that there are no two queens in the same column; the second that there are no two queens on the same diagonal. The program above can easily be adapted to solve the general n-queens problem. □

Confluence

The confluence test for ω_t described in (Abdennadher 1997) easily extends towards $\omega_t^\vee$. We extend the concept of joinability in the natural way towards $\omega_t^\vee$ states, which are (multi-)sets of ω_t states. Since different alternatives cannot influence each other under $\omega_t^\vee$, we only need to consider derivations starting in $\omega_t^\vee$ states consisting of a single alternative. The following theorem corresponds to Theorem 3.9 in (Abdennadher 1997) about local confluence under CHR's ω_t semantics.

Theorem 4 *A* CHR$^\vee$ *program is locally confluent under the* $\omega_t^\vee$ *semantics iff all its critical pairs are joinable.*

Proof: We extend the proof of Theorem 3.9 in (Abdennadher 1997) with cases for combinations of the **Derive** transition (i.e., one of the ω_t transitions), the **Split** transition, and the **Drop** transition. If the **Drop** transition applies in a

given state, it remains applicable in any state derived from it, and therefore all these states are joinable. The **Split** transition does not limit the applicability of the **Derive** transition and vice versa, so joinability is retained for combinations of these two transitions. The same holds for the combination of two **Split** transitions. The remaining cases consist of combinations of the **Derive** transition, which are dealt with by the proof in (Abdennadher 1997). □

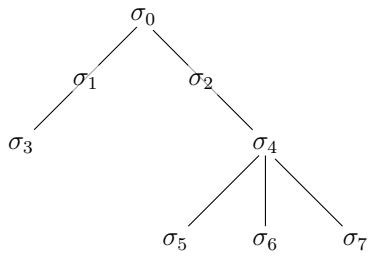
Finally, we can reuse the result from (Abdennadher 1997) that a terminating and *locally* confluent CHR$^{\vee}$ program is also confluent.

Search trees

An $\omega_t^{\vee}$ derivation can be visualized as a search tree. Such a search tree consists of a set of nodes and a set of (directed) edges connecting these nodes. A node is either an *internal* node or a *leaf* node. An internal node represents a choice point and corresponds to a state in which a **Split** transition applies. The *root* node corresponds to the initial state $\langle G, \emptyset, \texttt{true}, \emptyset \rangle_1$ with G the initial goal. It can be considered an internal node corresponding to a choice point with only one alternative. A leaf node represents a successful or failed final ω_t execution state. An edge goes from one node, its *start* node, to another node, its *end* node, and represents the derivation that transforms one of the alternatives of its start node into its end node, i.e., it consists of a series of execution states that are linked by **Derive** transitions. For example, the derivation

$$\{\sigma_0\} \overset{\omega_t^{\vee}}{\rightarrowtail}_P \{\sigma_1, \sigma_2\} \overset{\omega_t^{\vee}}{\rightarrowtail}_P \{\sigma_3, \sigma_2\} \overset{\omega_t^{\vee}}{\rightarrowtail}_P \{\sigma_3, \sigma_4\} \overset{\omega_t^{\vee}}{\rightarrowtail}_P \{\sigma_3, \sigma_5, \sigma_6, \sigma_7\}$$

corresponds to the following search tree:



In this example case, the search tree is traversed in left-to-right, depth-first order. We call the order in which search tree nodes are traversed, the *exploration order*. Note that different search trees are possible for the same goal. Consider for example the initial goal $\{(G_1 \vee G_2), (G_3 \vee G_4)\}$. For this goal, some derivations apply the **Split** transition to the subgoal $G_1 \vee G_2$ first, while others apply this transition first to $G_3 \vee G_4$.

4.3 Combining CHR^{rp} and CHR$^{\vee}$

In this section, we combine CHR^{rp} with CHR$^{\vee}$ into a flexible framework for defining both the exploration and propagation strategy to be used by the CHR constraint solver.

First, we discuss some issues concerning exploration strategies and constrained optimization. An exploration strategy determines the order in which solutions to a problem are generated. This order is only relevant if we need a subset of all solutions, in particular if we only need one. If we need all solutions, then a simple sorting of these solutions suffices to implement any exploration strategy. We assume here that search tree branches are processed independently, but refer to Section 4.5 for a discussion of conflict-directed backjumping, a search technique that does take into account results from other branches. In the case of naive optimization, implemented by computing all solutions and then choosing the optimal one amongst these, the exploration strategy is of no importance. However, in a more intelligent form, using for example a *branch & bound* approach, a good exploration strategy may cause considerable pruning of the search tree.

4.3.1 An Intermediate Step: CHR$^{rp}_{\vee}$

As an intermediate step, we introduce a simple combination of CHR^{rp} with CHR$^{\vee}$ into the combined language CHR$^{rp}_{\vee}$. This language supports execution control with respect to conjuncts by means of the CHR^{rp} rule priorities, but leaves the search control undetermined. The syntax of CHR$^{rp}_{\vee}$ is similar to that of CHR^{rp}, but also allows disjunction in the rule bodies, like in CHR$^{\vee}$. The operational semantics of CHR$^{rp}_{\vee}$ is almost the same as that of CHR$^{\vee}$. The only difference is that in the **Derive** transition, ω_t is replaced by ω_p. In the next subsection, we extend CHR$^{rp}_{\vee}$ with *branch priorities* to support the specification of exploration strategies. The resulting language is called CHR$^{brp}_{\vee}$. Finally, in Section 4.3.3, a correspondence result is given, relating CHR$^{brp}_{\vee}$ programs and derivations to CHR$^{\vee}$ programs and derivations.

4.3.2 Extending CHR$^{rp}_{\vee}$ with Branch Priorities: CHR$^{brp}_{\vee}$

Syntax

The syntax of CHR$^{brp}_{\vee}$ extends the syntax of CHR$^{rp}_{\vee}$ with *branch priorities*. A CHR$^{brp}_{\vee}$ simpagation rule looks as follows:

$$(bp, rp) :: r \ @ \ H^k \setminus H^r \iff g \mid bp_1 :: B_1 \vee \ldots \vee bp_m :: B_m$$

where r, H^k, H^r and g are as defined in Section 2.2.1. The *rule priority* rp is as in CHR^{rp}. The *branch priority* bp is a term: there is no a priori reason to restrict the allowed terms. In the examples, we use (tuples of) (lists of) integers and variables.

The rule body consists of a set of disjuncts B_i, each of which is annotated with a branch priority bp_i ($1 \leq i \leq m$). To simplify the presentation (e.g., of the correspondence result in Section 4.3.3), we impose that each disjunct B_i is a conjunction of constraints. In particular, we do not support nested disjunctions. If in a rule with a single disjunct, the branch priority of this disjunct equals the one of its parent branch, then the branch priorities can be omitted, and so each CHR$^{\text{rp}}$ rule is also a syntactically valid CHR$^{\text{brp}}_{\vee}$ rule.

A CHR$^{\text{brp}}_{\vee}$ program is a tuple $\langle \mathcal{R}, \mathcal{BP}, bp_0, \preceq \rangle$ where $\mathcal{R}$ is a set of CHR$^{\text{brp}}_{\vee}$ *rules*, $\mathcal{BP}$ is the *domain* of the branch priorities, $bp_0 \in \mathcal{BP}$ is the *initial branch priority*, and $\preceq$ is a *total preorder* relation over elements of $\mathcal{BP}$, which is to be defined in the host language.[1]

Operational semantics

We extend the ω_p states with a branch priority. The combination is called an *alternative* and is denoted by $bp :: \sigma$ where bp is the branch priority and σ is an ω_p execution state.[2] An alternative can be *marked*, in which case it is written as $bp :: \sigma^\star$. Marking is used to discard a solution in order to derive a next one.

The operational semantics $\omega_p^\vee$ considers (multi-)sets of alternatives. A total preorder $\preceq$ must be defined on their branch priorities, so that an alternative with the highest priority can be determined. In practice, we most often use a total order and in the examples, we define $\preceq$ by a logical formula containing arithmetic expressions. This implies that certain parts of the branch priorities must be ground. For a set Σ of alternatives, we denote by $\mathsf{max\_bp}(\Sigma)$ the highest branch priority of any unmarked alternative in Σ, or in case Σ does not contain any unmarked alternatives, a branch priority that is smaller than any other branch priority. The transitions of $\omega_p^\vee$ are given in Table 4.2.

The main differences w.r.t. the $\omega_t^\vee$ semantics are the following. The **Derive** transition is split up into three transitions corresponding to the ω_p transitions in order to support matching with, and guards involving the branch priority. These three transitions, as well as the **Split** and **Drop** transitions, only apply to the highest priority unmarked alternative. Finally, a new transition called **Mark** is introduced, whose purpose is to mark a solution (successful final state) in order to find a next solution. Given a goal G, we construct an initial $\omega_p^\vee$ state $\Sigma_0 = \{bp_0 :: \langle G, \emptyset, \mathtt{true}, \emptyset \rangle_1\}$ where bp_0 is the initial branch priority for the program.

Noteworthy is that the branch priorities presented above, do not change the shape of the search tree. Instead, they only influence the order in which the nodes of this search tree are explored. Therefore, this *exploration strategy* is only important in the following cases:

[1] We do not require that $\preceq$ is antisymmetric, i.e., $x \preceq y \wedge y \preceq x$ does not imply $x = y$.

[2] In the following, we treat states in which the goal contains a disjunction as ω_p states, but we do not call them final states, even if no ω_p transition applies to them.

1a. Solve $\{bp :: \langle\{c\} \uplus G, S, B, T\rangle_n\} \uplus \Sigma \overset{\omega_p^\vee}{\mapsto}_P \{bp :: \langle G, S, c \wedge B, T\rangle_n\} \uplus \Sigma$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and c is a built-in constraint.
1b. Introduce $\{bp :: \langle\{c\} \uplus G, S, B, T\rangle_n\} \uplus \Sigma \overset{\omega_p^\vee}{\mapsto}_P \{bp :: \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}\} \uplus \Sigma$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and c is a CHR constraint.
1c. Apply $\{bp :: \langle\emptyset, H_1 \uplus H_2 \uplus S, B, T\rangle_n\} \uplus \Sigma \overset{\omega_p^\vee}{\mapsto}_P \{bp :: \langle C, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n\} \uplus \Sigma$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and where P contains a rule of priority rp of the form $$(bp', rp) :: r \ @\ H_1' \backslash H_2' \iff g \mid C$$ and a matching substitution θ exists such that $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $bp = \theta(bp')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, $\theta(rp)$ is a ground arithmetic expression and $t = \langle \mathsf{id}(H_1), \mathsf{id}(H_2), r\rangle \notin T$. Furthermore, no rule of priority rp' and substitution θ' exists with $\theta'(rp') < \theta(rp)$ for which the above conditions hold.
2. Split $\{bp :: \sigma\} \uplus \Sigma \overset{\omega_p^\vee}{\mapsto}_P \{bp_1 :: \sigma_1, \ldots, bp_m :: \sigma_m\} \uplus \Sigma$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and where $\sigma = \langle\{bp_1 :: G_1 \vee \ldots \vee bp_m :: G_m\} \uplus G, S, B, T\rangle_n$ and $\sigma_i = \langle G_i \uplus G, S, B, T\rangle_n$ for $1 \leq i \leq m$.
3. Drop $\{bp :: \langle G, S, B, T\rangle_n\} \uplus \Sigma \overset{\omega_t^\vee}{\mapsto}_P \Sigma$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and $\mathcal{D} \models \neg\bar{\exists}_\emptyset B$.
4. Mark $\{bp :: \sigma\} \uplus \Sigma \overset{\omega_p^\vee}{\mapsto}_P \{bp :: \sigma^\star\} \uplus \Sigma$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and no other $\omega_p^\vee$ transition applies.

Table 4.2: Transitions of $\omega_p^\vee$

- if we require a subset of all solutions;
- if we combine the exploration strategy with an intelligent backtracking technique such as conflict-directed backjumping (see Section 4.5);
- if we need an optimal solution using a branch & bound or restart optimization approach.

In contrast, if we require all solutions and do not apply any pruning based on previously computed solutions, then the exploration strategy is irrelevant.[3]

In general, we might be interested in retrieving solutions one at a time. This is for example supported in the Prolog context by means of the toplevel environment asking whether more solutions are needed. We define a function $\mathsf{find\_next}$ that returns, given an $\omega_p^\vee$ state, the first solution in this state, as well as the resulting state after marking this solution, which contains the remaining solutions.

$$\mathsf{find\_next}(\Sigma) = \langle A, \Sigma_R \rangle$$

if there exists a derivation $\Sigma \overset{\omega_p^\vee}{\longmapsto}{}^*_P \Sigma_A \overset{\omega_p^\vee}{\longmapsto}_P \Sigma_R$ where the transition from Σ_A to Σ_R is a **Mark** transition and the derivation from Σ to Σ_A does not contain such a transition. In the result, $A = B \wedge \mathsf{chr}(S)$ where $bp :: \langle \emptyset, S, B, T \rangle_n$ is the highest priority unmarked alternative in Σ_A.

Constrained optimization

As a general approach to constrained optimization, we show how both a *branch & bound* and *restart optimization* scheme can be implemented in CHR$^{\mathrm{brp}}_\vee$. We consider a goal G whose *best* solution is to be returned. This best solution is such that there exists no solution that assigns a lower value to a given *cost* function F whose variables appear in G.[4] Let there be given an initial $\omega_p^\vee$ state Σ_0 based on the goal G. Under the assumption that there is a solution, we find the solution that minimizes our cost function F as $\mathsf{find\_min}(\Sigma_0, F)$ where $\mathsf{find\_min}$ is defined as

$$\mathsf{find\_min}(\Sigma, F) = \quad \text{let} \begin{cases} \langle A, \Sigma' \rangle = \mathsf{find\_next}(\Sigma) \\ \Sigma'' = \mathsf{add\_goal}(F < F(A), \Sigma') \end{cases} \quad \text{in} \quad \begin{array}{l} \text{if } \Sigma'' \text{ has no solution then } A \\ \text{else } \mathsf{find\_min}(\Sigma'', F) \end{array}$$

[3] This is related to the distinction between variable ordering and value ordering in case all solutions are required, see for example (Smith and Sturdy 2005).

[4] Alternative, one can use a *utility* function that is to be maximized.

for branch & bound optimization, and for restart optimization as follows:

$$\mathsf{find\_min}(\Sigma, F) = \quad \text{let} \begin{cases} \langle A, \Sigma' \rangle = \mathsf{find\_next}(\Sigma) \\ \Sigma'' = \mathsf{add\_goal}(F < F(A), \Sigma) \end{cases} \\ \text{in} \quad \text{if } \Sigma'' \text{ has no solution then } A \\ \text{else } \mathsf{find\_min}(\Sigma'', F)$$

Here $\mathsf{add\_goal}(G, \Sigma)$ returns the $\omega_p^{\vee}$ state that results from adding the goal G to all unmarked alternatives in Σ:

$$\mathsf{add\_goal}(G, \Sigma) = \{bp :: \sigma^\star \mid bp :: \sigma^\star \in \Sigma\} \uplus \\ \{bp :: \langle G \uplus G', S, B, T \rangle_n \mid bp :: \langle G', S, B, T \rangle_n \in \Sigma\}$$

The difference between both approaches is that in branch & bound optimization, the new bound is added as a goal to all remaining alternatives, whereas in restart optimization, the new bound is added to the initial state, which is solved for again.

Dynamic search tree generation

It is not always convenient or even possible to state at compile time which alternatives are to be created. In general, we may want to generate alternatives dynamically, at runtime. Let us assume we can compute, by means of a built-in constraint, a list of all alternatives to be generated, as well as a list of priorities to be used for the respective alternatives. We can then dynamically generate the alternatives by asserting a `generate_alternatives`/2 constraint whose arguments are respectively the list of priorities and the list of alternatives. Let $\top \in \mathcal{BP}$ be higher than all branch priorities used for the dynamically generated alternatives. We can then define the `generate_alternatives`/2 constraint by the rules of Listing 4.2. By using the $\top$ priority value, we ensure that all alternatives are generated before the first of them is processed.

```
(_,1) :: generate_alternatives([P|Ps],[A|As]) <=>
                              P :: A ∨ ⊤ :: generate_alternatives(Ps,As).
(_,1) :: generate_alternatives([],[]) <=> true.
```

Listing 4.2: Dynamic search tree generation

4.3.3 Correspondence between $\omega_p^{\vee}$ and $\omega_t^{\vee}$ Derivations

In this subsection, we show that every $\omega_p^{\vee}$ derivation of a CHR$_{\vee}^{\text{brp}}$ program P, corresponds to an $\omega_t^{\vee}$ derivation of a corresponding non-deterministic CHR$^{\vee}$ version of this program P. We first show how to create such a non-deterministic version

of a CHR$^{\mathrm{brp}}_{\vee}$ program. Next, we propose a mapping from $\omega_p^{\vee}$ states to $\omega_t^{\vee}$ states, and finally, we give the correspondence result.

Given a CHR$^{\mathrm{brp}}_{\vee}$ program P, we create a CHR$^{\vee}$ program $\mathsf{nondet}(P)$ as follows. For every rule in P of the form

$$(bp, rp) :: r \ @ \ H^k \setminus H^r \iff \mathit{guard} \mid bp_1 :: B_1 \vee \ldots \vee bp_m :: B_m$$

$\mathsf{nondet}(P)$ contains a rule

$$r \ @ \ H^k \setminus H^r, \mathtt{bp}(bp) \iff \mathit{guard} \mid (\mathtt{bp}(bp_1), B_1) \vee \ldots \vee (\mathtt{bp}(bp_m), B_m)$$

and for every rule of the form

$$(bp, rp) :: r \ @ \ H^k \setminus H^r \iff \mathit{guard} \mid \mathit{body}$$

where *body* does not contain a disjunction, $\mathsf{nondet}(P)$ contains a rule

$$r \ @ \ \mathtt{bp}(bp), H^k \setminus H^r \iff \mathit{guard} \mid \mathit{body}$$

It is easy to see that given one $\mathtt{bp}/1$ constraint in the initial goal, no state can be derived in which the CHR constraint store contains more than one such constraint. We incorporate the branch priorities of P as constraints into $\mathsf{nondet}(P)$ because they may appear in guards or rule bodies. In the following, we assume that no constraint identifier is used for the $\mathtt{bp}/1$ constraint and that it is ignored by the propagation history. This is necessary to prevent that a propagation rule instance can fire again when the branch priority changes. We can achieve it by means of a source-to-source transformation in which constraint identifiers and the propagation history are made explicit.

Now we define a mapping function map from $\omega_p^{\vee}$ states to $\omega_t^{\vee}$ states.

$$\begin{aligned}
\mathsf{map}(\{bp :: \sigma\} \uplus \Sigma) &= \mathsf{map}(bp :: \sigma) \uplus \mathsf{map}(\Sigma) \\
\mathsf{map}(bp :: \langle G, S, B, T\rangle_n) &= \begin{cases} \langle G', S, B, T\rangle_n & \text{if } G \text{ consists of a disjunction} \\ \langle G, \{\mathtt{bp}(bp)\} \cup S, B, T\rangle_n & \text{otherwise} \end{cases} \\
&\quad \text{where } G' \text{ is found by replacing each disjunct} \\
&\quad bp_i :: B_i \text{ in } G \text{ by a disjunct } (\mathtt{bp}(bp_i), B_i)
\end{aligned}$$

Theorem 5 (Correspondence) *Given a CHR$^{brp}_{\vee}$ program P and the corresponding CHR$_{\vee}$ program $\mathsf{nondet}(P)$, then for each transition $\Sigma \overset{\omega_p^{\vee}}{\longmapsto}_P \Sigma'$, there exists a derivation $\mathsf{map}(\Sigma) \overset{\omega_t^{\vee}}{\longmapsto}^*_{\mathsf{nondet}(P)} \mathsf{map}(\Sigma')$ and if Σ is a final $\omega_p^{\vee}$ state, then $\mathsf{map}(\Sigma)$ is a final $\omega_t^{\vee}$ state.*

Proof: Let there be given a transition $\Sigma \overset{\omega_p^{\vee}}{\longmapsto}_P \Sigma'$. Each $\omega_p^{\vee}$ transition operates on a single (highest priority, unmarked) alternative $bp :: \sigma$, and replaces this

alternative with one or more new alternatives. Let $\Sigma = \{bp :: \sigma\} \uplus \Sigma_R$ and let $\Sigma' = \Sigma_N \uplus \Sigma_R$. We also have that $\mathsf{map}(\Sigma) = \mathsf{map}(bp :: \sigma) \uplus \mathsf{map}(\Sigma_R)$. Now a transition from Σ to Σ' can be one of the following:

1a. Solve $\sigma = \langle \{c\} \uplus G, S, B, T\rangle_n$ with c a built-in constraint, and so $\mathsf{map}(bp :: \sigma) = \langle \{c\} \uplus G, \{\mathsf{bp}(bp)\} \cup S, B, T\rangle_n$. Therefore, in $\mathsf{map}(\Sigma)$, the $\omega_t^{\vee}$ **Derive** transition is possible since the ω_t **Solve** transition applies to $\mathsf{map}(bp :: \sigma)$. This results in the $\omega_t^{\vee}$ state $\{\langle G, \{\mathsf{bp}(bp)\} \cup S, c \wedge B, T\rangle_n\} \uplus \mathsf{map}(\Sigma_R)$ which is exactly $\mathsf{map}(\{bp :: \langle G, S, c \wedge B, T\rangle_n\} \uplus \Sigma_R)$.

1b. Introduce $\sigma = \langle \{c\} \uplus G, S, B, T\rangle_n$ with c a CHR constraint, and so $\mathsf{map}(bp :: \sigma) = \langle \{c\} \uplus G, \{\mathsf{bp}(bp)\} \cup S, B, T\rangle_n$. Therefore, in $\mathsf{map}(\Sigma)$, the $\omega_t^{\vee}$ **Derive** transition is possible since the ω_t **Introduce** transition applies to $\mathsf{map}(bp :: \sigma)$. This results in the $\omega_t^{\vee}$ state $\{\langle G, \{c\#n, \mathsf{bp}(bp)\} \cup S, B, T\rangle_{n+1}\} \uplus \mathsf{map}(\Sigma_R)$ which is exactly $\mathsf{map}(\{bp :: \langle G, \{c\#n\} \cup S, c \wedge B, T\rangle_{n+1}\} \uplus \Sigma_R)$.

1c. Apply $\sigma = \langle \emptyset, H_1 \uplus H_2 \uplus S, B, T\rangle_n$ and so $\mathsf{map}(bp :: \sigma) = \langle \emptyset, \{\mathsf{bp}(bp)\} \uplus H_1 \uplus H_2 \uplus S, B, T\rangle_n$. P contains a rule of the form

$$(bp', rp) :: r \ @ \ H_1' \backslash H_2' \iff g \mid C$$

and there exists a matching substitution θ such that $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $bp = \theta(bp')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, $\theta(rp)$ is a ground arithmetic expression and $t = \langle \mathsf{id}(H_1), \mathsf{id}(H_2), r\rangle \notin T$. Assume C consists of a disjunction, then $\mathsf{nondet}(P)$ contains a rule of the form

$$r \ @ \ H_1' \backslash H_2', \mathsf{bp}(bp') \iff g \mid C'$$

where C' is found by replacing all CHR$_{\vee}^{\mathrm{brp}}$ disjuncts of the form $bp_i :: B_i$ by CHR$^{\vee}$ disjuncts of the form $(\mathsf{bp}(bp_i), B_i)$. Given this rule, the ω_t **Apply** transition applies to state $\mathsf{map}(bp :: \sigma)$. In particular, we can use θ as the matching substitution. Therefore, the $\omega_t^{\vee}$ **Derive** transition applies to state $\mathsf{map}(\Sigma)$, resulting in the state $\{\langle C', H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n\} \uplus \mathsf{map}(\Sigma_R)$ where $t = \langle \mathsf{id}(H_1), \mathsf{id}(H_2), r\rangle$. This corresponds to the state $\mathsf{map}(\Sigma') = \mathsf{map}(\{bp :: \langle C, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n\} \uplus \Sigma_R)$. The case that C does not contain a disjunction is similar. The difference is that the $\mathsf{bp}/1$ constraint is not removed and the rule body remains unchanged.

2. Split $\sigma = \langle \{bp_1 :: G_1 \vee \ldots \vee bp_m :: G_m\}, S, B, T\rangle_n$ and so $\mathsf{map}(bp :: \sigma) = \langle \{(\mathsf{bp}(bp_1), G_1) \vee \ldots \vee (\mathsf{bp}(bp_m), G_m)\}, S, B, T\rangle_n$. In $\mathsf{map}(\Sigma)$, the $\omega_t^{\vee}$ **Split** transition applies, resulting in the state $\{\langle \{\mathsf{bp}(bp_1)\} \uplus G_1, S, B, T\rangle_n, \ldots, \langle \{\mathsf{bp}(bp_m)\} \uplus G_m, S, B, T\rangle_n\} \uplus \mathsf{map}(\Sigma_R)$. In each of the m first alternatives, the **Introduce** transition applies, introducing the $\mathsf{bp}/1$ constraint into the CHR constraint store. After these introductions, the resulting state equals $\{\langle G_1, \{\mathsf{bp}(bp_1)\} \cup S, B, T\rangle_n, \ldots, \langle G_m, \{\mathsf{bp}(bp_m)\} \cup S, B, T\rangle_n\} \uplus$

$\mathsf{map}(\Sigma_R)$. This state equals $\mathsf{map}(\Sigma')$ since $\Sigma' = \{bp_1 :: \langle G_1, S, B, T\rangle_n, \ldots, bp_m :: \langle G_m, S, B, T\rangle_n\} \uplus \Sigma_R)$.

3. Drop $\sigma = \langle G, S, B, T\rangle_n$ and $\mathcal{D} \models \neg\bar{\exists}_\emptyset B$. Since the map function does not change the built-in constraint store, the $\omega_t^\vee$ **Drop** transition applies to $\mathsf{map}(\Sigma)$ resulting in the state $\mathsf{map}(\Sigma_R)$. The result of applying the **Drop** transition to Σ is the state Σ_R and so the resulting states correspond.

4. Mark This transition does not change the result of the map function and so $\mathsf{map}(\Sigma) = \mathsf{map}(\Sigma')$.

This proves the first part of the theorem. For the second part, consider a final $\omega_p^\vee$ state Σ. Such a state consists of a set of alternatives, each of which is marked. An alternative is marked only if no other transition applies. This means (amongst others) that the goal of such an alternative is empty and its built-in constraint store is consistent. Once an alternative is marked, it remains unchanged. Now consider the $\omega_t^\vee$ state $\mathsf{map}(\Sigma)$. If an $\omega_t^\vee$ transition applies to this state, then this must be a **Derive** transition because the map function only adds a $\mathtt{bp}/1$ constraint to the CHR constraint store of each alternative with an empty goal and so the **Split** and **Drop** transitions are not applicable. Let σ be the alternative in $\mathsf{map}(\Sigma)$ that is replaced by a **Derive** transition. Since the goal of σ is empty, the ω_t transition corresponding to the **Derive** transition must be an **Apply** transition, firing a rule of the form

$$r \ @ \ H_1' \backslash H_2', \mathtt{bp}(bp') \iff g \mid C'$$

Let $\theta(r)$ be the fired rule instance, then the CHR constraint store of state σ must contain sets of constraints H_1 and H_2 matching the heads H_1' and H_2', as well as a constraint $\mathtt{bp}(bp)$ that matches with $\mathtt{bp}(bp')$. Moreover, the built-in constraint store of σ entails the guard g in conjunction with the matching substitution θ. Now, let $bp :: \sigma'$ be the $\omega_p^\vee$ alternative in Σ that maps on ω_t state σ. By definition, the CHR constraint store of σ' contains the constraints in H_1 and H_2, and its built-in constraint store and propagation history are equal to the ones of σ. The CHR$_\vee^{\mathrm{brp}}$ program P contains a rule

$$(bp', rp) :: r \ @ \ H_1' \backslash H_2' \iff g \mid C$$

for which holds that rule instance $\theta(r)$ can fire given the branch priority, CHR and built-in constraint stores and propagation history of $\omega_p^\vee$ alternative $bp :: \sigma'$. Potentially, $\theta(r)$ is not the highest priority applicable rule instance in $bp :: \sigma'$, but then another rule instance can fire, and so this also implies that Σ is not a final $\omega_p^\vee$ state. So we conclude that a non-final $\omega_t^\vee$ state corresponds to a non-final $\omega_p^\vee$ state, which proves the second part of the theorem. □

4.4 Specifying Common Exploration Strategies

In this section, we show how different exploration strategies can be implemented in $\text{CHR}_\vee^{\text{brp}}$. In Section 4.4.1, we look at uninformed strategies such as depth-first, breadth-first and depth-first iterative deepening. It is shown that a $\text{CHR}_\vee^{\text{rp}}$ program (i.e., one without branch priorities) can be automatically translated into a $\text{CHR}_\vee^{\text{brp}}$ program that implements these exploration strategies. Next, in Section 4.4.2, we consider informed exploration strategies such as best-first search, A* and limited discrepancy search. Finally, in Section 4.4.3, we show how different strategies can be combined, with as an example a mixture of depth- and breadth-first search.

4.4.1 Uninformed Exploration Strategies

Depth-first and breadth-first search

In order to implement depth-first or breadth-first search, we transform each $\text{CHR}_\vee^{\text{rp}}$ rule of the form

$$rp :: H^k \setminus H^r \iff guard \mid B_1 \vee \ldots \vee B_n$$

into a $\text{CHR}_\vee^{\text{brp}}$ rule

$$(D, rp) :: H^k \setminus H^r \iff guard \mid (D+1) :: B_1 \vee \ldots \vee (D+1) :: B_n$$

The branch priorities correspond to the depth in the search tree: $\mathcal{BP} = \mathbb{N}$ and $bp_0 = 0$. We define the branch priority order $\preceq$ as follows:

- for depth-first search, $D_1 \preceq D_2 \Leftrightarrow D_1 \leq D_2$
- for breadth-first search, $D_1 \preceq D_2 \Leftrightarrow D_1 \geq D_2$

Now, the branch priorities are such that for depth-first search, the deeper alternative has a higher priority, whereas for breadth-first search, the more shallow alternative has a higher priority.

Example 4.2 (4-queens (ctd.)) The 4-queens solver given in Listing 4.1 can be extended with branch and rule priorities as shown in Listing 4.3.

The branch priorities implement depth-first or breadth-first search, depending on the $\preceq$ order used. The rule priorities ensure that (further) labeling is done only *after* consistency checking. The derivation starts with the following initial $\text{CHR}_\vee^{\text{brp}}$ state: $\{0 :: \langle\{\texttt{queens}\}, \emptyset, \texttt{true}, \emptyset\rangle_1\}$. □

The implementation of depth- and breadth-first search given above is still nondeterministic with respect to alternatives at equal depth. We can implement a deterministic *left-to-right* version of depth-first or breadth-first search as follows.

```
(_,1) :: queens <=> row(1), row(2), row(3), row(4).
(D,2) :: row(R) <=> (D+1) :: queen(R,1) ∨ (D+1) :: queen(R,2) ∨
                                (D+1) :: queen(R,3) ∨ (D+1) :: queen(R,4).
(_,1) :: queen(_ ,C1), queen(_ ,C2) ==> C1 =\= C2.
(_,1) :: queen(R1,C1), queen(R2,C2) ==> abs(R1 - R2) =\= abs(C1 - C2).
```

Listing 4.3: CHR$^{\text{brp}}_{\vee}$ implementation of 4-queens

Take as branch priorities sequences of integers ($\mathcal{BP} = \mathbb{N}^*$ and $bp_0 = \epsilon$). The length of the sequence denotes the depth in the search tree, and the i^{th} element in the sequence denotes the number of the branch taken at level i.

The order over these priorities is defined as

$$(D, rp) :: H^k \setminus H^r \iff \mathit{guard} \mid (D +\!\!+ [1]) :: B_1 \vee \ldots \vee (D +\!\!+ [n]) :: B_n$$

with

$$L_1 \succeq L_2 \Leftrightarrow (\mathsf{length}(L_1) > \mathsf{length}(L_2)) \vee \big(\mathsf{length}(L_1) = \mathsf{length}(L_2) \wedge L_1 \leq^d L_2\big)$$

for depth-first search and

$$L_1 \succeq L_2 \Leftrightarrow (\mathsf{length}(L_1) < \mathsf{length}(L_2)) \vee \big(\mathsf{length}(L_1) = \mathsf{length}(L_2) \wedge L_1 \leq^d L_2\big)$$

for breadth-first search. Here $\leq^d$ is the lexicographic or *dictionary* order.

Depth-limited search and depth-first iterative deepening

Depth-limited search is a variant of depth-first search in which search tree nodes are only expanded up to a given depth bound. It is an incomplete search in that it is not able to find solutions beyond this depth bound. Amongst others, depth-limited search is used in iterative deepening search. It can be implemented in CHR$^{\text{brp}}_{\vee}$ by using the depth-first search program given in the previous paragraph, extended with the following rule:

```
(D,1) :: limit(D) <=> false.
```

This rule ensures that any alternative at the depth limit fails. Its rule priority ensures the rule is tried before any other rule.[5] Here, the depth limit is given by an appropriate `limit`/1 constraint which is to be added to the initial goal.

Depth-first iterative deepening (see e.g. (Korf 1985)) consists of iteratively running depth-limited search, increasing the depth limit in each run. Iterative deepening can be implemented by adding the following rule:

[5] This may require increasing the rule priorities of all other rules by one.

```
(_,1) :: deepen(D) <=> 1 :: limit(D) ∨ 0 :: deepen(D+1).
```

Instead of a `limit`/1 constraint, the goal is extended with a `deepen(1)` constraint. Using the above approach may lead to an infinite loop in which the depth limit keeps increasing in case the search tree is finite but contains no solutions. The reason is that it is not possible to distinguish between failure because the depth limit is reached, and failure because the entire search tree has been traversed and no solutions were found. In Section 4.5.4 we return to this issue and show how conflict-directed backjumping can solve this problem. More precisely, it is shown that if failure is independent of the depth limit, there is no need to change it.

Iterative broadening

Iterative broadening (Ginsberg and Harvey 1992) works similar to iterative deepening, but instead of using a depth limit that is iteratively increased, the number of branches starting at any given node is limited, and this limit increases over the iterations. The domain of branch priorities $\mathcal{BP} = \{\langle D, B\rangle \mid D, B \in \mathbb{N}\}$. A node with branch priority $\langle D, B\rangle$ is at depth D in the search tree, and is the B^{th} alternative of its parent node. We use as initial branch priority $bp_0 = \langle 0, 1\rangle$ and we define the $\preceq$ relation as follows:

$$\langle D_1, \_\rangle \preceq \langle D_2, \_\rangle \Leftrightarrow D_1 \leq D_2$$

Now, the code of Listing 4.4 implements iterative broadening.

```
(_,1) :: broaden(B) <=> B < n_max | ⟨1,1⟩ :: limit(B) ∨ ⟨0,1⟩ :: broaden(B+1).

(⟨_,B⟩,1) :: limit(L) <=> L < B | false.

(⟨D,_⟩,rp) :: r @ H^k \ H^r <=> guard | ⟨D+1,1⟩ :: B_1 ∨ ... ∨ ⟨D+1,n⟩ :: B_n.
```

Listing 4.4: Iterative broadening in $\text{CHR}^{\text{brp}}_{\vee}$

In the code, $n_{\texttt{max}}$ is an upperbound on the number of alternatives in a rule body, and is used to ensure termination. The depth-first strategy ensures that the subtree for a given breadth limit is completely traversed before increasing this breadth limit (by means of a `broaden`/1 constraint). The second component of the branch priorities of the alternatives created by the first rule, is of no importance, as long as it is less than the breadth limit. We extend the initial goal with a `broaden(1)` constraint to start the process.

4.4.2 Informed Exploration Strategies

Informed strategies take problem dependent heuristics into account.

Limited and depth-bounded discrepancy search

The limited discrepancy search (LDS) (Harvey and Ginsberg 1995) strategy is similar to best-first search. It is designed for problems with binary choices only, in which the alternatives for each choice are ordered according to some heuristic. The idea behind LDS is that if following the heuristic does not work, then it is probable that a solution can be found by violating the heuristic only a limited number of times. Each violation of the heuristic is called a discrepancy, and the algorithm consists of first trying those alternatives with at most one discrepancy, then the ones with at most two discrepancies and so on until a solution is found. Let there be given the following CHR$^{rp}_{\vee}$ labeling rule:

```
rp :: domain(X,[V1,V2]) <=> (X = V1) ∨ (X = V2).
```

where the values in the domain are ordered according to the heuristic, i.e., $\mathtt{V}_1$ is preferred over $\mathtt{V}_2$. In CHR$^{brp}_{\vee}$, we can write this labeling rule as follows:

```
(D,rp) :: domain(X,[V1,V2]) <=> D :: (X = V1) ∨ (D+1) :: (X = V2).
```

The branch priority represents the number of discrepancies: $\mathcal{BP} = \mathbb{N}$. Initially, there is no discrepancy: $bp_0 = 0$. In each choice point, the discrepancy remains unchanged if we follow the heuristic, and is increased by one if we deviate from this heuristic. The alternatives with fewer discrepancies are explored first: $P_1 \preceq P_2 \Leftrightarrow P_1 \geq P_2$.

Note that the description of LDS in (Harvey and Ginsberg 1995) uses depth-first search combined with a limit on the number of discrepancies. Here, we use a form of best-first search where alternatives with less discrepancies are preferred. A variant of LDS in which the number of discrepancies is actually bounded, can be expressed in the same way as how we express depth-limited search, i.e., by introducing a `limit`/1 constraint and adding a rule

```
(D,1) :: limit(D) <=> false.
```

Another variant of LDS, called *depth-bounded discrepancy search* (DDS) (Walsh 1997) combines LDS with iterative deepening. It is best characterized as a depth-first search that only allows discrepancies below a certain depth, that is iteratively increased. As for depth-first search, the branch priority denotes the depth in the search tree ($\mathcal{BP} = \mathbb{N} \cup \{-1\}$) and we start at the root ($bp_0 = 0$). The order over these priorities is the same as for depth-first search. Listing 4.5 shows the result.

The `limit`/1 constraint represents the current depth limit. Above this limit any number of discrepancies are allowed (`allow_discrepancy`/0), while below the limit no discrepancies are allowed (`no_discrepancy`/0). It gets complicated when we are at the current depth limit. Let us first focus on the iterative deepening

```
(_,1) :: start <=>
             0 :: no_discrepancy ∨ -1 :: (allow_discrepancy, deepen(0)).
(_,1) :: deepen(D) <=> 0 :: limit(D) ∨ -1 :: deepen(D+1).
(D,1) :: limit(D), allow_discrepancy <=> force_discrepancy.

(D,rp) :: allow_discrepancy \ domain(X,[V1,V2]) <=>
                                    (D+1) :: (X = V1) ∨ (D+1) :: (X = V2).
   rp  :: force_discrepancy, domain(X,[_,V2]) <=> no_discrepancy, X = V2.
   rp  :: no_discrepancy \ domain(X,[V1,_]) <=> X = V1.
```

Listing 4.5: Depth-bounded discrepancy search in $\mathrm{CHR}_{\vee}^{\mathrm{brp}}$

part. The `deepen`/1 constraint drives the iterative loop of installing successively increasing depth limits. The extra element $-1 \in \mathcal{BP}$ is the minimal priority, which ensures that we only install the next depth limit after the current one has been fully explored. Each successive iteration should only produce additional solutions, which have not been found in preceding iterations. Hence, all solutions should exploit the increased depth-limit and have a discrepancy at that depth. The `force_discrepancy`/0 constraint makes sure this happens. By adding the `start`/0 constraint to the goal, we get the process going.

Similar to the case of depth-first iterative deepening, the depth limit keeps increasing if either the problem is overconstrained, or we require all solutions. Again, using conflict-directed backjumping remedies this problem. Only if failure happens in a state in which the constraint store contains a `force_discrepancy`/0 or `no_discrepancy`/0 constraint (justified by a `limit`/1 constraint), the depth limit is changed.

A* and iterative deepening A*

The A* algorithm (Hart et al. 1968) consists of using best-first search to find an optimal solution in a constrained optimization problem. Let the branch priorities be such that $p :: \sigma$ is better than $p' :: \sigma'$ for successful final states σ and σ', if and only if $p \succeq p'$; and $\{p :: \sigma\} \overset{\omega_p^{\vee}}{\rightarrowtail}{}_P^* \Sigma$ implies $p \succeq p_i$ for all $p_i :: \sigma_i \in \Sigma$. Under these conditions, the first solution found (using find_next) is also an optimal solution.

Example 4.3 (Shortest path distance) The rules of Listing 4.6 compute the shortest path distance between two nodes in a directed graph.

The `neighbors`/2 constraints represent the edges of the graph: for each node v, there exists a `neighbors`(v, CU) constraint with CU a list containing a pair $c - u$ for each edge from node v to node u with cost c. The initial goal consists of a single `path`/2 constraint whose arguments are the nodes for which we wish to compute the shortest path distance. The branch priorities denote the distance

```
(_,1) :: path(V,V) <=> true.
(D,2) :: neighbors(V,CU) \ path(V,W) <=> 1 :: branches(W,D,CU).

(_,1) :: branches(W,D,[C-U|CUs]) <=>
                              (D+C) :: path(U,W) ∨ 1 :: branches(W,D,CUs).
(_,1) :: branches(_,_,[]) <=> false.
```

Listing 4.6: $\text{CHR}^{\text{brp}}_{\vee}$ implementation of a shortest path algorithm

between the initial start node and the node represented by the first argument of the `path`/2 constraint currently in the store, or, eventually in a successful final state, the distance between the initial start and end nodes. We have $\mathcal{BP} = \mathbb{N}$, $bp_0 = 0$ and $D_1 \preceq D_2 \Leftrightarrow D_1 \geq D_2$. □

The A* algorithm normally also uses a heuristic function. In the above example, we have used 0 as a heuristic underestimate of the remaining cost to the various end nodes. Better estimates are possible; however, we have chosen to keep the example simple.

Iterative deepening A* (ID-A*) is a combination of A* and depth-first iterative deepening. It consists of depth-first search in part of the search tree, only considering nodes whose cost function does not exceed a certain threshold value. If no solution is found, the threshold is increased by the minimal amount needed to include an unseen search tree node. We can easily implement a variant of ID-A* in which the threshold is increased by some fixed amount, similar to how we implement depth-first iterative deepening. However, increasing the threshold with the minimal amount needed to include the lowest cost unseen node, falls outside of the scope of our framework as it requires communication between different branches of the search tree.

4.4.3 Combining Basic Exploration Strategies

Now we show that the $\text{CHR}^{\text{brp}}_{\vee}$ language is expressive enough to formulate complex exploration strategies. In particular, with an appropriate choice of priorities and orderings, compositions of the previous exploration strategies can be expressed. In this subsection, we give two examples of such strategy compositions.

Example 4.4 Consider we want to use breadth-first search, but only up to a certain depth, e.g. so as not to exceed available memory. Beyond the depth limit, the search should switch to a depth-first strategy.

To implement this more complex strategy, we use the same branch priorities as the ones used for depth-first and breadth-first search in Section 4.4.1. The following definition for the $\preceq$ relation is used

$$D_1 \preceq D_2 \Leftrightarrow (D_2 \leq T \wedge D_2 \leq D_1) \vee (T \leq D_1 \leq D_2)$$

where T is the depth threshold which is given by the user. In words, beyond the threshold, the deeper alternative is preferred, whereas below the threshold, the more shallow alternative is preferred. An alternative whose depth is below the threshold is preferred over one whose depth is beyond the threshold.

□

In the second example, we show how to traverse the states resulting from some **Split** transitions in a depth-first order, and others in a breadth-first order. Such a strategy allows us for instance to do a breadth-first search globally (to ensure completeness), with some subcomputations (that are known to terminate) done using depth-first search.

Example 4.5 Let there be given a $\text{CHR}^{\text{rp}}_{\vee}$ program containing the following two binary labeling rules:

```
1 :: label_df @ df_domain(X,[V1,V2]) <=> X = V1 ∨ X = V2.
1 :: label_bf @ bf_domain(X,[V1,V2]) <=> X = V1 ∨ X = V2.
```

Furthermore assume we want the labeling by rule `label_df` to proceed in depth-first, and the labeling by rule `label_bf` in breadth-first order.

We define a *depth-first subtree* as part of the search tree in which all internal nodes correspond to depth-first splits. A *breadth-first* subtree is similarly defined. Comparing alternatives within a depth-first or breadth-first subtree is straightforward. For alternatives across such subtrees, we proceed as follows.

Let there be two alternatives A_1 and A_2, let $N^1 = [N^1_1, \ldots, N^1_{n_1}]$ be the sequence of nodes on the unique path from the root of the search tree, to (but excluding) alternative A_1, and let $N^2 = [N^2_1, \ldots, N^2_{n_2}]$ be the sequence of nodes on the path from the root to alternative A_2. Let $i_1 \in \{1, \ldots, n_1 + 1\}$ be the first index for which it holds that node N_{i_1} corresponds to a breadth-first (depth-first) split or $n_1 + 1$ if no such index exists. In particular, the nodes in the subsequence $[N^1_1, \ldots, N^1_{i_1-1}]$ all correspond to depth-first (breadth-first) splits. Let $i_2 \in \{1, \ldots, n_2 + 2\}$ be similarly defined. If $i_1 > i_2$ ($i_1 < i_2$) then alternative A_1 has priority over alternative A_2 and if $i_1 < i_2$ ($i_1 > i_2$) then the opposite holds. Finally, if $i_1 = i_2$, we compare the depths of the first depth-first (breadth-first) splits in sequences $[N^1_{i_1}, \ldots, N^1_{n_1}]$ and $[N^2_{i_2}, \ldots, N^2_{n_2}]$ and so on until either the depths are different or there are no more nodes to consider, in which case both alternatives have an equal priority.

Figure 4.1 shows an example search tree including its depth-first and breadth-first subtrees. Node 2 (as well as all alternatives on the edge from Node 1 to Node 2) has priority over Node 1 as both are part of the same depth-first subtree and Node 2 is deeper than Node 1. Node 1 has priority over Nodes 3 and 4, even though the latter are deeper in the global search tree. The reasoning is as follows. The depth of the first breadth-first split in the path from the root node to respectively

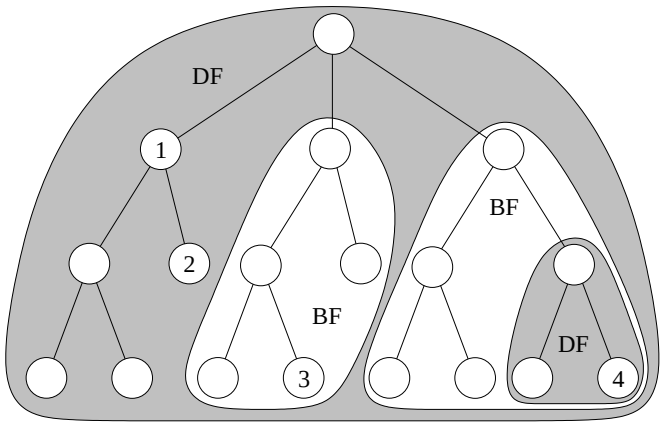


Figure 4.1: Search tree for mixed depth-and-breadth-first search

Nodes 1, 3 and 4 all equals 2 (for Node 1, there is no such split). In the remaining subpaths (for Node 1, this path is empty), the depth of the first depth-first split equals 1 for Node 1, 3 for Node 3 and 2 for Node 4. Therefore, Node 1 has priority over Nodes 3 and 4. Furthermore, Node 4 has priority over Node 3. Finally, by transitivity, Node 2 also has priority over Nodes 3 and 4.

In CHR$^{brp}_{\vee}$, we can model the above described preference relation using the following rules.[6]

```
(P,1) :: label_df @ df_domain(X,[V1,V2]) <=>
          df_child_priority(P) :: X = V1 ∨ df_child_priority(P) :: X = V2.
(P,1) :: label_bf @ bf_domain(X,[V1,V2]) <=>
          bf_child_priority(P) :: X = V1 ∨ bf_child_priority(P) :: X = V2.
```

where the functions `df_child_priority` and `bf_child_priority` are implemented as follows:

```
df_child_priority(Parent) = Child :-
        (   length(Parent) mod 2 = 0
        ->  Child = Parent ++ [2]
        ;   append(Context,[Depth],Parent),
            Child = Context ++ [Depth + 1]
        ).

bf_child_priority(Parent) = Child :-
        (   length(Parent) mod 2 = 1
        ->  Child = Parent ++ [2]
```

[6] We use Mercury syntax here (Somogyi et al. 1996), which supports a functional notation for predicates; $p(\bar{X}, X_n)$:– *body* is equivalent to $p(\bar{X}) = X_n$:– *body*.

```
    ;   append(Context,[Depth],Parent),
        Child = Context ++ [Depth + 1]
    ).
```

The initial branch priority is set to ϵ. We define the order relation $\succeq$ as follows:

$$L_1 \succeq L_2 \Leftrightarrow L_1 \succeq^d L_2$$

where

$$[H_1|T_1] \succeq^d \epsilon$$
$$[H_1|T_1] \succeq^d [H_2|T_2] \Leftrightarrow H_1 > H_2 \vee (H_1 = H_2 \wedge T_1 \succeq^b T_2)$$

and

$$\epsilon \succeq^b [H_2|T_2]$$
$$[H_1|T_1] \succeq^b [H_2|T_2] \Leftrightarrow H_1 < H_2 \vee (H_1 = H_2 \wedge T_1 \succeq^d T_2)$$

The branch priority of an alternative is a list of depths of the first depth-first or breadth-first split in consecutive subpaths from the root to the alternative as defined earlier.[7] For example, in Figure 4.1, the branch priority equals [2] for Node 1, [3] for Node 2, $[2, 3]$ for Node 3, and $[2, 2, 2]$ for Node 4. It is assumed that the root node is a depth-first node; if necessary, a dummy depth-first split can be created to ensure this is true. One can now verify that

$$[3] \succeq [2] \succeq [2, 2, 2] \succeq [2, 3]$$

□

4.5 Look Back Schemes: Conflict-Directed Backjumping

Wolf et al. (2007) use *justifications* and *conflict sets* to define an *extended and refined operational semantics* $\omega_r^{\vee\star}$, which supports *look back* schemes like conflict-directed backjumping (CBJ) (Prosser 1993) and dynamic backtracking (Ginsberg 1993) as opposed to standard chronological backtracking. In this section, we show how their approach can be combined with our framework. More precisely, we propose an extended version of the $\omega_p^{\vee}$ semantics that supports conflict-directed backjumping, and discuss the correctness and optimality of this extension. We note that the benefits of CBJ are limited when strong constraint propagation and a good variable ordering heuristic is used (see for example (Chen and van Beek 2001)). We have chosen not to support dynamic backtracking, as it requires changing the shape of the search tree, which may conflict with the execution order imposed by the rule priorities. Moreover, for efficiency it requires an adaptive version of CHR (Wolf et al. 2000).

[7] The list is implicitly assumed to end with an infinite sequence $\mathsf{ones} = [1|\mathsf{ones}]$.

4.5.1 Justifications and Labels

Justifications are introduced in the CHR context by Wolf et al. (2000) and used for the purpose of finding *no-goods* in conflict-directed backjumping by Wolf et al. (2007). In that context, justifications consist of the *choices* that caused a given constraint to hold. The ω_p semantics can easily be extended to support justifications by annotating each constraint in the goal, CHR constraint store, and built-in constraint store with a justification. A constraint c with justification J is denoted by c^J and we write $\mathsf{just}(c^J) = J$. This notation is extended towards (multi-)sets of constraints in the obvious way. The transitions of the extended ω_p semantics are given as part of the extended priority semantics of CHR$^{\mathrm{brp}}_{\vee}$ in the next subsection.

In case of depth-first search, conflicts can be uniquely described using only the search tree levels at which the conflicting choices were made. This is the approach taken in (Wolf et al. 2007). For more general exploration strategies, we need a more precise specification of the choices involved in a conflict. Therefore, we introduce a labeling scheme for search tree nodes that allows us to distinguish between such nodes, and to decide whether a given node is a descendant of another. In general, we can use as labels the branch priorities used in the (deterministic) left-to-right versions of depth- and breadth-first search proposed in Section 4.4.1.

Finally, assume all descendants of a given search tree node N fail, say with justifications $J_1, \ldots, J_n$. Each of these justifications is a set of choices that together lead to failure. Now a new justification J for the failure of node N can be created by merging $J_1, \ldots, J_n$: $J = (J_1 \cup \ldots \cup J_n) \setminus D$ where D consists of the labels of descendants of N.

4.5.2 The Extended Priority Semantics of CHR$^{\mathbf{brp}}_{\vee}$

In analogy with (Wolf et al. 2007), we extend the $\omega_p^{\vee}$ semantics into the $\omega_p^{\vee\star}$ semantics, whose states are tuples $\langle K, S\rangle$ where K is the *conflict set*, a set of *no-goods*, i.e., justifications of failures; and S is a set of (marked and unmarked) ω_p states extended with a branch priority and branch label: $S = \{bp_1 :: bl_1 \ @ \ \sigma_1, \ldots, bp_n :: bl_n \ @ \ \sigma_n\}$. By slight abuse of notation, we use bl to refer to just the label, or to the labeled alternative $bp :: bl \ @ \ \sigma$, depending on the context.

While in case of depth-first search, backjumping consists of skipping a series of alternatives, in general, it requires pruning of the search tree. This is because the alternatives that are skipped with depth-first search are exactly the (remaining) children of the deepest conflict, while in general, these children may be scheduled for later resolution when using a different exploration strategy and hence they cannot be skipped, but must be explicitly removed.

The transitions of the extended priority semantics of CHR$^{\mathrm{brp}}_{\vee}$ are given in Table 4.3. The **Backjump** transition implements what corresponds to a multi-step backjump in (Wolf et al. 2007). It works by constructing a new failed alternative as a child of the node to which is jumped back. If this alternative is the only

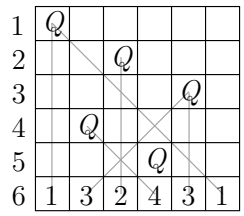


Figure 4.2: Partial solution to the 6-queens problem

remaining child of its parent, then the next applicable transition will again be a **Backjump** transition. Otherwise, a **Backtrack** transition will follow, which just removes the failed alternative. The latter is treated as a special case in (Wolf et al. 2007), by the single-step backjump transition.

Example 4.6 (6-queens) Consider the 6-queens problem where we are labeling the last row and all previously rows have their queens set as in Figure 4.2.

We see that all positions in the last row are conflicting with some previously set queen. For each position, we have given the *first* row which contains a conflicting queen. Note that some positions conflict with more than one queen: for example for the second column, these are the queens in rows 3 and 4. Because the fifth row participates in none of the conflicts, we can jump back to the queen in the fourth row and move her, instead of considering alternative places for the queen in the fifth row. In $\mathrm{CHR}_{\vee}^{\mathrm{brp}}$, we can use the solver shown in Listing 4.7.

```
   1  :: queens <=> row(1), ..., row(6).
(D,7) :: row(R) <=> (D+1) :: queen(R,1) ∨ ... ∨ (D+1) :: queen(R,6).
   R₁ :: queen(R₁,C₁), queen(_ ,C₂) ==> C₁ =\= C₂.
   R₁ :: queen(R₁,C₁), queen(R₂,C₂) ==> abs(R₁ - R₂) =\= abs(C₁ - C₂).
```

Listing 4.7: Solver for 6-queens with preference for early conflicts

This solver differs from the one in Listing 4.3 in that conflicts with queens in early rows are preferred above those with queens in later rows. This preference is imposed by using the row number of the conflicting queen as rule priority.[8] It ensures that we jump back as far as possible.

In the remainder of this example, we use a simplified notation for $\omega_p^{\vee\star}$ states. More precisely, we represent an alternative of the form $bp :: bl \mathbin{@} \langle G, S, B, T\rangle_n$ as $bp :: \mathsf{chr}(S) \cup B$, i.e., we do not show the branch label (which is assumed to be equal to the branch priority), goal, propagation history or next free identifier, and we represent CHR constraints without their identifier. We assume a depth-first

[8] A conflict between the queens in rows i and j ($i < j$) can be detected by a rule instance of priority i, and a symmetric one of priority j. Only the first one fires.

1a. Solve $\langle K, \{bp :: bl \mathbin{@} \langle \{c^J\} \uplus G, S, B, T\rangle_n\} \uplus \Sigma\rangle \overset{\omega_p^{\vee\star}}{\longmapsto}_P \langle K, \{bp :: bl \mathbin{@} \langle G, S, c^J \wedge B, T\rangle_n\} \uplus \Sigma\rangle$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and c is a built-in constraint.

1b. Introduce $\langle K, \{bp :: bl \mathbin{@} \langle \{c^J\} \uplus G, S, B, T\rangle_n\} \uplus \Sigma\rangle \overset{\omega_p^{\vee\star}}{\longmapsto}_P \langle K, \{bp :: bl \mathbin{@} \langle G, \{c^J \# n\} \cup S, B, T\rangle_{n+1}\} \uplus \Sigma\rangle$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and c is a CHR constraint.

1c. Apply $\langle K, \{bp :: bl \mathbin{@} \langle \emptyset, H_1 \uplus H_2 \uplus S, B, T\rangle_n\} \uplus \Sigma\rangle \overset{\omega_p^{\vee\star}}{\longmapsto}_P \langle K, \{bp :: bl \mathbin{@} \langle C^J, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n\} \uplus \Sigma\rangle$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and where P contains a rule of priority rp of the form

$$(bp', rp) :: r \mathbin{@} H_1' \backslash H_2' \iff g \mid C$$

and a matching substitution θ exists such that $\mathsf{chr}(H_1) = \theta(H_1')$, $\mathsf{chr}(H_2) = \theta(H_2')$, $bp = \theta(bp')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, $\theta(rp)$ is a ground arithmetic expression and $t = \langle \mathsf{id}(H_1), \mathsf{id}(H_2), r\rangle \notin T$. Furthermore, no rule of priority rp' and substitution θ' exists with $\theta'(rp') < \theta(rp)$ for which the above conditions hold. The justification $J = \mathsf{just}(H_1) \cup \mathsf{just}(H_2) \cup \mathsf{just}(E)$ where E is a minimal subset of B for which holds that $\mathcal{D} \models E \rightarrow \bar{\exists}_E(\theta \wedge g)$.

2. Split $\langle K, \{bp :: bl \mathbin{@} \sigma\} \uplus \Sigma\rangle \overset{\omega_p^{\vee\star}}{\longmapsto}_P \langle K, \{bp_1 :: bl_1 \mathbin{@} \sigma_1, \ldots, bp_m :: bl_m \mathbin{@} \sigma_m\} \uplus \Sigma\rangle$ where $\sigma = \langle (bp_1 :: G_1 \vee \ldots \vee bp_m :: G_m)^J \wedge G, S, B, T\rangle_n$, $\mathsf{max\_bp}(\Sigma) \preceq bp$, and $\sigma_i = \langle G_i^{J \cup \{bl_i\}} \wedge G, S, B, T\rangle_n$ for $1 \leq i \leq m$.

3a. Backtrack $\langle K, \{bp :: bl \mathbin{@} \langle G, S, B, T\rangle_n\} \uplus \Sigma\rangle \overset{\omega_p^{\vee\star}}{\longmapsto}_P \langle K \cup \{J\}, \Sigma\rangle$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$, $\mathcal{D} \models \neg\bar{\exists}_\emptyset B$, and there exists at least one alternative in Σ that is a descendant of the parent of bl. Here, J is the justification of the inconsistency of B.

3b. Backjump $\langle K, \{bp :: bl \mathbin{@} \sigma\} \uplus \Sigma\rangle \overset{\omega_p^{\vee\star}}{\longmapsto}_P \langle K \backslash K', \{bp :: bl' \mathbin{@} \langle \emptyset, \emptyset, \mathtt{false}^J, \emptyset\rangle_1\} \uplus \Sigma'\rangle$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$, σ is a failed ω_p state, and there exists no other alternative in Σ that is a descendant of the parent of bl. Let K' be the justifications in K that involve bl or one of its siblings. A new no-good J is created by merging these justifications, removing the labels of bl and its siblings. Now let bl' be the deepest alternative in J. We find Σ' by removing from Σ all descendants of bl'.

4. Mark $\langle K, \{bp :: bl \mathbin{@} \sigma\} \uplus \Sigma\rangle \overset{\omega_p^{\vee\star}}{\longmapsto}_P \langle K, \{bp :: bl \mathbin{@} \sigma^\star\} \uplus \Sigma\rangle$ if $\mathsf{max\_bp}(\Sigma) \preceq bp$ and no other $\omega_p^{\vee\star}$ transition applies.

Table 4.3: Transitions of $\omega_p^{\vee\star}$

strategy which implies that it is sufficient to use the depth in the search tree as branch labels. Now using our simplified representation, the $\omega_p^{\vee\star}$ state right before labeling the last row looks as follows:

$$\langle K, \{(5 :: Q_5 \cup \{\mathtt{row(6)}^{\emptyset}\})\} \cup \Sigma\rangle$$

where $Q_4 = \{\mathtt{queen(1,1)}^{\{1\}}, \mathtt{queen(2,3)}^{\{2\}}, \mathtt{queen(3,5)}^{\{3\}}, \mathtt{queen(4,2)}^{\{4\}}\}$, $Q_5 = Q_4 \cup \{\mathtt{queen(5,4)}^{\{5\}}\}$ and $\Sigma = \{(5 :: Q_4 \cup \{\mathtt{queen(5,5)}^{\{5\}}, \mathtt{row(6)}^{\emptyset}\}), (5 :: Q_4 \cup \{\mathtt{queen(5,6)}^{\{5\}}, \mathtt{row(6)}^{\emptyset}\})\} \cup \Sigma'$. The contents of K and Σ' is not relevant to our presentation. The labeling rule replaces this $\omega_p^{\vee\star}$ state by

$$\langle K, \{(6 :: Q_5 \cup \{\mathtt{queen(6,1)}^{\{6\}}\}), \ldots, (6 :: Q_5 \cup \{\mathtt{queen(6,6)}^{\{6\}}\})\} \cup \Sigma\rangle$$

Now each of the queens on row 6 conflicts with a queen in an earlier row. These conflicts lead to failures that are justified by the conflicting constraints' labels. After having dealt with columns 1 to 5, the resulting $\omega_p^{\vee\star}$ state is

$$\langle K \cup \{\{1,6\},\{3,6\},\{2,6\},\{4,6\},\{3,6\}\}, \{(6 :: Q_5 \cup \{\mathtt{queen(6,6)}^{\{6\}}\})\} \cup \Sigma\rangle$$

So far, we have only used **Backtrack** transitions to deal with failures. The last alternative position on the sixth row again leads to failure, this time with justification $\{1,6\}$. Now, the **Backjump** transition applies, which forms a new justification by merging the ones involving the sixth row. This new justification equals $\{1,2,3,4\}$. The **Backjump** transition removes all (two) remaining alternatives on the fifth row and creates a new failed alternative, resulting in the state

$$\langle K, \{(6 :: \mathtt{false}^{\{1,2,3,4\}})\} \cup \Sigma'\rangle$$

after which (in this case) a **Backtrack** transition follows and the next alternative on row 4 is tried:

$$\langle K \cup \{\{1,2,3,4\}\}, \Sigma'\rangle$$

□

4.5.3 Correctness and Optimality Issues

We now discuss three issues concerning the correctness and optimality of conflict-directed backjumping in $\mathrm{CHR}^{\vee}$ (and $\mathrm{CHR}_{\vee}^{(\mathrm{b})\mathrm{rp}}$). Firstly, for correctness, we need to impose restrictions on the programs for which we can use conflict-directed backjumping. In particular, we require that a program is confluent with respect to the $\omega_t^{\vee}$ semantics. The following theorem states the correctness of the $\omega_p^{\vee\star}$ semantics.

Theorem 6 *For a given program P whose non-deterministic version $\mathsf{nondet}(P)$ is confluent with respect to the $\omega_t^{\vee}$ semantics, it holds that $\Sigma_0 \overset{\omega_p^{\vee\star}}{\longmapsto}{}_P^* \Sigma_n$ with Σ_n a final $\omega_p^{\vee\star}$ state, if and only if $\Sigma_0 \overset{\omega_p^{\vee}}{\longmapsto}{}_P^* \Sigma_n$.*

Proof (sketch): The main proof obligation consists of showing that the **Backjump** transition is correct, i.e., that it does not discard any solutions. We show that this is true given that the existence of a failing $\omega_t^\vee$ derivation for a state $\mathsf{map}(\Sigma)$ in the non-deterministic version $\mathsf{nondet}(P)$ of a program P, implies that all such derivations fail.

Consider a node N, all of whose children have failed, where the failures are justified by justifications $J_1, \ldots, J_n$. A justification J_i $(1 \leq i \leq n)$ is a set of branch labels. After the last child of N fails, a **Backjump** transition applies under $\omega_p^{\vee^\star}$. This transition consists of merging the justifications $J_1, \ldots, J_n$ into a new justification J by uniting the respective sets of branch labels, discarding the labels of children of N.

Let bl be the branch label of the deepest alternative in J. The branch labeled bl ends in a node, i.e., a state in which a **Split** transition applies (see Section 4.2), which is preceded by an **Apply** transition, firing some rule instance $\theta(r)$. Let $bp :: bl \mathbin{@} \sigma$ be the alternative right before $\theta(r)$ fired. In this alternative and under the $\omega_t^\vee$ semantics, we can also fire the rule instance that lead to the creation of node N, say $\theta'(r')$, in state $\mathsf{map}(bp :: \sigma)$ using program $\mathsf{nondet}(P)$ (see Section 4.3.3). The reasoning is that none of the children of N depend on constraints derived after state σ as their branch labels would otherwise have been part of the justification J.

Now there exists a failing derivation $D = \{\mathsf{map}(bp :: \sigma)\} \overset{\omega_t^\vee}{\longmapsto}{}^*_{\mathsf{nondet}(P)} \emptyset$ which consists of first firing rule instance $\theta'(r')$ and then repeating the derivations that lead to the failure of each of the children of node N. If we have that the existence of a failing derivation such as D, implies that all derivations starting in the same state also fail, then we also have that all children of the ending node of branch bl fail under the $\omega_p^{\vee^\star}$ semantics, and so we can safely discard them as is done in the **Backjump** transition. We call the above property *conflict preservation*. It is a notion weaker than confluence in the sense that all solutions should only be the same in case one of them is a failure. In practice we can use the notion of *confluence*, for which a decidable test exists in case of a terminating program (Abdennadher 1997),[9] to decide whether or not we can apply conflict-directed backjumping to a program. Figure 4.3 shows how the failing $\omega_t^\vee$ derivation D can be constructed, starting in state $\mathsf{map}(bp :: \sigma)$, and justifying the backjump. □

The following example shows how solutions can be missed in case a program is not confluent.

Example 4.7 Consider the CHR$^{\text{rp}}_\vee$ program below, with initial goal {g}.

```
1 :: a, e <=> false.
1 :: a, f <=> false.
1 :: a, d <=> true.
```

[9] This test is defined for the ω_t semantics, but can easily be extended towards the $\omega_t^\vee$ semantics.

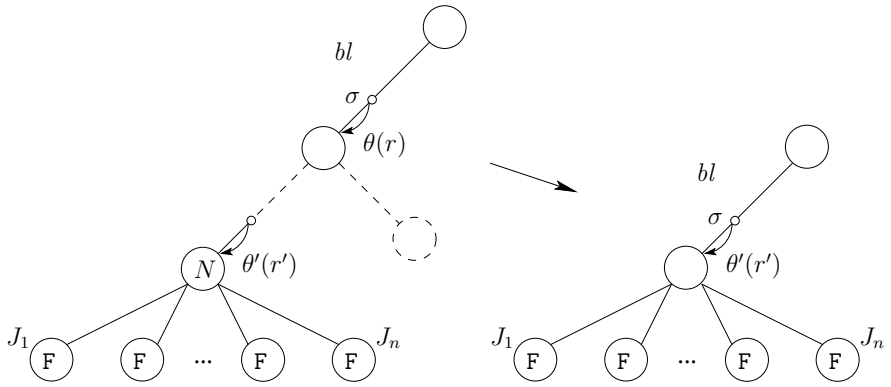


Figure 4.3: Construction of a failing derivation, justifying a backjump.

```
2 :: g ==> a ∨ b.
3 :: g ==> c ∨ d.
4 :: g ==> e ∨ f.
```

We have left out branch priorities as they are not needed for the example. Assume a first labeling step chooses `a`, then `c` is chosen, and finally both `e` and `f` fail due to `a` and hence regardless of the second choice point (in which `c` was chosen). However, we cannot jump back to the first choice, because if we choose `d` in the second choice point, the `a` constraint is 'consumed' and both `e` and `f` are consistent. The problem is that the program is not confluent w.r.t. the $\omega_t^\vee$ semantics. For example, if the constraint store in a given state consists of the CHR constraints `a`, `e` and `d`, then the result can be either a failed state, or a state in which the constraint store contains the constraint `e`. □

A second issue is concerned with optimality. In general, when it is detected that a constraint is in conflict with some other constraints, there might be other sets of conflicting constraints as well. In the conflict-directed backjumping algorithm, it is assumed that testing for conflicts follows the order in which the variables are instantiated. In general, we prefer conflicts with constraints that appear closer to the root of the search tree.

In the 6-queens program of Listing 4.7, we dealt with this issue by giving conflicts with earlier queens a higher (rule) priority. This approach easily extends towards other finite domain constraint solvers that do not apply look-ahead schemes. However, in general, a conflict may involve multiple CHR and built-in constraints, each of which has its own justification. Therefore, it might not always be clear which rule instance is preferred. We could for example minimize the depth of the deepest choice, the number of choices, or the sum of the depths of

the choices involved in a conflict. Some strategies maximize the depth of the first backjump, while others maximize for example the total number of nodes skipped.

A final issue is that choice points should preferably be created only after all constraint propagation and consistency checking is completed. Otherwise, it is possible that a failure is independent of the last choice made. By assigning a low rule priority to labeling rules, we reach consistency before further labeling.

4.5.4 Iterative Deepening Revisited

In Section 4.4.1 the iterative deepening strategy is implemented. However, as noted, there is a problem with termination. In iterative deepening, the depth limit should only increase if the search tree contains an unexplored subtree. So, when a depth-limited search fails, we are interested in why it fails. This type of information is not available in the $\omega_p^\vee$ semantics, but it is available in the $\omega_p^{\vee\star}$ semantics by means of the conflict set.

When a failure occurs because the depth limit is reached, this failure is due to the following rule:

```
(D,1) :: limit(D) <=> false.
```

The justification for the failure hence contains the justification of the `limit`/1 constraint. If however no failure occurs because the depth limit is reached, then the justification for the failure does not contain the one of the `limit`/1 constraint and therefore, the second alternative of the rule that generated this constraint, namely

```
(_,1) :: deepen(D) <=> 1 :: limit(D) ∨ 0 :: deepen(D+1).
```

will be pruned by the **Backjump** transition.

Noteworthy is that the above approach only works if branch priorities are annotated with justifications themselves, and these justifications are taken into account when constraints rely on the branch priority. To simplify the presentation, we have ignored this in our description of the $\omega_p^{\vee\star}$ semantics in Section 4.5.2. We note that a constraint, asserted after firing some rule instance $\theta(r)$, should contain the justification of the branch priority only if either the firing of $\theta(r)$ depends on the branch priority, or the branch priority is used in the arguments of the constraint. If we do not take into account the justifications of the branch priority, then it may happen that a backjump is made to change the depth limit if all children of some node fail due to this depth limit, without taking into account that other nodes may be below the depth limit.

4.6 Related Work

Adding different search strategies to declarative languages, and in particular Constraint (Logic) Programming languages, has been done before. For a more thor-

ough overview, see for example (Frühwirth et al. 2006; Schulte and Carlsson 2006). In (Schulte et al. 1994) it is shown how search tree states can be encapsulated in the multi-paradigm language Oz. The encapsulation is implemented using a variable scoping mechanism and local versions of the constraint store, and allows the implementation of different search strategies. A generalization of this work is presented for the functional logic programming language Curry in (Hanus and Steiner 1998). An important aspect there is that the encapsulated search tree states are evaluated lazily. The encapsulation of search tree states allows them to be reordered and modified freely, and in particular, communication between different alternatives is feasible. Such communication is currently not supported by our framework.

Much related to our work is the OPL modeling language (van Hentenryck et al. 2000) in which search strategies consist of three parts: the *exploration strategy*, the *limit strategy* and the *selection strategy*. The exploration strategy consists of an evaluation function that assigns a score to search tree nodes, much like our branch priorities, combined with a procedure to decide when to switch to another node. The limit strategy imposes bounds on the time and space used for searching. Finally, the selection strategy determines which solutions are returned and can be used for instance to implement constrained optimization. Our framework also supports the definition of exploration strategies. Limit strategies are supported to a certain extent: we have shown how to implement limits on the search tree depth and breadth, as well as on the discrepancy w.r.t. some heuristic (as part of limited discrepancy search). However, we do not support limits on time or space usage. Finally, we showed how constrained optimization can be implemented, which is an example of a selection strategy. Other selection strategies can be implemented in a similar fashion.

Constraint Logic Programming systems such as ECL^{i}PS^e (Wallace et al. 1997) or SICStus Prolog's `clp(fd)` (Carlsson et al. 1997) are usually limited by Prolog's built-in depth-first search mechanism.[10] However, using language features such as the `findall`/3 predicate or blackboard primitives, other strategies can be implemented. For example, in Ciao Prolog (Hermenegildo et al. 1996), breadth-first and depth-first iterative deepening search are supported using a source transformation. Bruynooghe (2004) uses the blackboard primitives of SICStus Prolog to implement intelligent backtracking.

In all of the above mentioned work, the search strategy is stated independently of the program logic and therefore treats all search in a uniform way. Our approach supports both uniform and rule specific strategies, thereby allowing the use of different search strategies for different parts of the program.

[10] Search strategies in these systems mostly consist of different ways of *shaping* the search tree, with the *exploration* strategy being fixed to left-to-right depth-first.

Related work in the CHR context

In the context of CHR$^{\vee}$, Menezes et al. (2005) propose a CHR$^{\vee}$ implementation for Java in which the search tree is made explicit and manipulated at runtime to improve efficiency. In particular, when a rule firing is independent of the choice made in a previous choice point, the result of this rule firing is valid for all alternatives in that choice point, and so by reordering the nodes in the search tree, some redundant work is avoided. In a sense, this reasoning takes into account *justifications* of constraints, i.e., those constraints that caused the derivation of any given constraint.

Justifications are introduced in CHR in the context of adaptive CHR (Wolf et al. 2000), which extends CHR by supporting the incremental adaption of CHR derivations in the context of both constraint insertions and deletions. In (Wolf 2005), justifications, in particular for the built-in constraint *false*, are used to implement intelligent search algorithms such as *conflict-directed backjumping* and *dynamic backtracking*. A new operational semantics for CHR$^{\vee}$, called the *extended and refined operational semantics* $\omega_r^{\vee\star}$, which formally combines the concept of justifications with CHR$^{\vee}$, is given in (Wolf et al. 2007). The semantics in fact implements conflict-directed backjumping. We extend the work of Wolf et al. (2007) by also supporting exploration strategies different from left-to-right depth-first in combination with conflict-directed backjumping, and by discussing optimality and correctness issues.

In (Robin et al. 2007), it is proposed to transform the disjuncts in CHR$^{\vee}$ into special purpose constraints which can be dealt with by an external search component. An example of such an external search component is the Java Abstract Search Engine (JASE), which is part of the Java Constraint Kit (JACK) (Abdennadher et al. 2002), a CHR implementation for Java. Since Java does not offer native search in contrast with Prolog, Java-based CHR implementations need to implement their own search support. In practice, this often means more flexibility compared to the built-in search in Prolog. In particular, Prolog only supports a limited way of movement between nodes of the search tree, whereas in JASE, one can jump from one search tree node to another by means of trailing, copying or recomputation, as well as combinations of these methods.

4.7 Conclusion

To conclude, we summarize our contributions. In Section 4.3 we combined and extended two language extensions of CHR, namely CHR^{rp} and CHR$^{\vee}$, into a flexible framework for execution control in CHR. In this framework, called CHR$_{\vee}^{brp}$, the propagation strategy is determined by means of *rule priorities*, whereas the exploration strategy is determined by means of *branch priorities*. In Section 4.4, we have shown how various tree exploration strategies can be expressed in our

framework. These strategies include *uninformed* exploration strategies such as depth-first, breadth-first and depth-first iterative deepening (Section 4.4.1), *informed* strategies such as limited discrepancy search (Section 4.4.2), as well as combinations of different strategies (Section 4.4.3). Finally, in Section 4.5, we have adapted the work of Wolf et al. (2007) which proposes a combination of conflict-directed backjumping (CBJ) with $\text{CHR}^\vee$, to our framework, by adding support for exploration strategies different from left-to-right depth-first. Moreover, we have established correctness and optimality conditions for this combination of CBJ with $\text{CHR}^{\text{brp}}_\vee$.

Future work

A first topic for future work is the (efficient) implementation of our search framework in CHR systems. In particular, it is worth considering adding such search support to systems that currently do not offer search facilities like the K.U.Leuven JCHR (Van Weert et al. 2005) and CCHR (Wuille et al. 2007) systems. Some issues that an optimized implementation should deal with are the use of specialized priority queues for e.g. depth-first and breadth-first search, and the choice between copying, trailing and recomputation (as well as combinations of these) for the task of jumping between search tree nodes (see also (Schulte 1999)).

It would also be interesting to have a high-level way to combine different exploration strategies. The approach we have taken in Section 4.4.3 is rather ad hoc, and in general it remains unclear what the priority domain, order relation, and priority assignments should look like.

Finally, we have already shown how information from successes (e.g., branch & bound in Section 4.3.2) and failures (e.g., conflict-directed backjumping in Section 4.5) in other branches of the search tree can be used to speed up the constraint solving process. It would be interesting to see how similar approaches can be used, for example to implement iterative deepening A^* or the dynamic variable ordering technique proposed by Müller (2005). The latter consists of changing the variable to be labeled next after a backjump, taking into account the reason for the backjump.

Chapter 5

Complexity Analysis of CHR^{rp} Programs

This chapter investigates the relationship between the Logical Algorithms language (LA) of Ganzinger and McAllester and Constraint Handling Rules. We present a translation schema from LA to CHR^{rp}, and show that the meta-complexity theorem for LA can be applied to a subset of CHR^{rp} via inverse translation. Inspired by the high-level implementation proposal for Logical Algorithms by Ganzinger and McAllester and based on a new scheduling algorithm, we propose an alternative implementation for CHR^{rp} that gives strong complexity guarantees and results in a new and accurate meta-complexity theorem for CHR^{rp}. It is furthermore shown that the translation from Logical Algorithms to CHR^{rp} combined with the new CHR^{rp} implementation, satisfies the required complexity for the Logical Algorithms meta-complexity result to hold.

5.1 Introduction

Recently, Sneyers et al. (2005) have shown that all algorithms can be implemented in CHR while preserving both time and space complexity.

In "Logical Algorithms" (LA) (Ganzinger and McAllester 2002) (and based on previous work in (Ganzinger and McAllester 2001; McAllester 1999)), a bottom-up logic programming language is presented for the purpose of facilitating the derivation of complexity results of algorithms described by logical inference rules. This problem is far from trivial because the runtime is not necessarily proportional to the derivation length (i.e., the number of rule applications), but also includes the cost of pattern matching for multi-headed rules, as well as the costs related to high-level execution control which is specified using rule priorities in the Logical Algorithms language. The language of Ganzinger and McAllester resembles CHR

in many ways and has often been referred to in the discussion of complexity results of CHR programs (Christiansen 2005; Frühwirth 2002b; Schrijvers and Frühwirth 2006; Sneyers et al. 2006a). In particular, Christiansen (2005) uses the meta-complexity theorem that accompanies the Logical Algorithms language, and notes that the CHR system used (SICStus CHR by Holzbaur and Frühwirth (1998)) does not always exhibit the right complexity because previously computed partial rule matches are not stored.

The aim of this chapter is to investigate the relationship between both languages. More precisely, we look at how the meta-complexity theorem for Logical Algorithms can be applied to (a subset of) CHR, and how CHR can be used to implement Logical Algorithms with the correct complexity. First, we present a translation schema from Logical Algorithms to CHR^{rp}. Logical Algorithms derivations of the original program correspond to CHR^{rp} derivations in the translation and vice versa. We also show how to translate a subclass of CHR^{rp} programs into Logical Algorithms. This allows us to apply the meta-complexity theorem for Logical Algorithms to these CHR^{rp} programs as well. Because the Logical Algorithms meta-complexity theorem is based on an optimized implementation, it gives more accurate results than the implementation independent meta-complexity theorem of Frühwirth (2002a, 2002b) while being more general than the ad-hoc complexity derivations in (Schrijvers and Frühwirth 2006; Sneyers et al. 2006a).

Our current implementation of CHR^{rp} as presented in Chapter 3 does not guarantee the complexity required for the meta-complexity theorem for Logical Algorithms to hold via translation to CHR^{rp}. Another issue is that the translation from CHR^{rp} to Logical Algorithms is restricted to a subset of CHR^{rp}. Therefore, we propose a new implementation of CHR^{rp}, designed such that it supports a new meta-complexity theorem for the complete CHR^{rp} language, while also ensuring that LA programs translated into CHR^{rp} are executed with the correct complexity. We note that this alternative implementation is not optimized for average case performance, but is designed to achieve certain complexity guarantees.

More precisely, the implementation is based on the high-level implementation proposal for Logical Algorithms as given by Ganzinger and McAllester (2002), and on a new scheduling data structure proposed for this purpose, and described in detail in (De Koninck 2007). The implementation is described by means of translation to regular CHR. By using a CHR system with advanced indexing support, such as the K.U.Leuven CHR system (Schrijvers and Demoen 2004), our implementation achieves the complexity required to enable a new and accurate meta-complexity result for the whole CHR^{rp} language.

Overview The rest of this chapter is organized as follows. In Section 5.2, the syntax and semantics of the Logical Algorithms language is reviewed and the known meta-complexity theorems for both LA and CHR are presented. In Section 5.3 a translation of LA programs to CHR^{rp} programs is presented and in

Section 5.4, the opposite is done for a subset of CHR^{rp}. Section 5.5 proposes an alternative implementation for CHR^{rp} which enables a new meta-complexity theorem for this language, given in Section 5.6. Some concluding remarks are given in Section 5.7.

5.2 Logical Algorithms and CHR^{rp}

In this section, we give an overview of the syntax and semantics of Logical Algorithms (Section 5.2.1) and review the meta-complexity results that are known for both LA and CHR (Section 5.2.2).

5.2.1 Logical Algorithms

This subsection gives an overview of the syntax and semantics of the Logical Algorithms language.

Syntax

A Logical Algorithms program $P = \{r_1, \ldots, r_n\}$ is a set of rules. Ganzinger and McAllester (2002) use a graphical notation to represent rules. We use a new textual representation that is closer to the syntax of CHR. A Logical Algorithms rule is an expression

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

where r is the rule *name*, the atoms A_i (for $1 \leq i \leq n$) are the *antecedents* and C is the *conclusion*, which is a conjunction of atoms whose variables appear in the antecedents. Rule r has *priority* p where p is an arithmetic expression whose variables (if any) occur in the first antecedent A_1. If p contains variables, then r is called a dynamic priority rule. Otherwise, it is called a static priority rule. In the graphical notation of Ganzinger and McAllester (2002), the above rule is represented as shown below.

$$(r, p) \quad \frac{\begin{array}{c} A_1 \\ \vdots \\ A_n \end{array}}{C}$$

The arguments of an atom are either Herbrand terms or (integer) arithmetic expressions. There are two types of atoms: comparisons and user-defined atoms. A comparison has the form $x < y$, $x \leq y$, $x = y$ or $x \neq y$ with x and y arithmetic expressions or, in case of $(=)/2$ and $(\neq)/2$, Herbrand terms. Comparisons are only allowed in the antecedents of a rule and all variables in a comparison must appear in earlier antecedents. A user-defined atom can be positive or negative. A

1. Apply $\sigma \overset{\text{LA}}{\mapsto}_P \sigma \cup \theta(C)$ if there exists a (renamed apart) rule r in P of priority p of the form

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

and a ground substitution θ such that for every antecedent A_i,

- $\mathcal{D} \models \theta(A_i)$ if A_i is a comparison
- $\theta(A_i) \in \sigma$ and $\texttt{del}(\theta(A_i)) \notin \sigma$ if A_i is a positive user-defined atom
- $\theta(A_i) \in \sigma$ if A_i is a negative user-defined atom

Furthermore, $\theta(C) \not\subseteq \sigma$ and no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold.

Table 5.1: The Logical Algorithms operational semantics

negative user-defined atom has the form $\texttt{del}(A)$ where A is a positive user-defined atom. A ground user-defined atom is called an assertion.

Example 5.1 An example rule (from Dijkstra's shortest path algorithm as presented in (Ganzinger and McAllester 2002)) with name `d2` and priority 1 is

```
d2 @ 1 : dist(V,D1), dist(V,D2), D2 < D1 => del(dist(V,D1)).
```

The antecedent $\texttt{D2 < D1}$ is a comparison, the atoms $\texttt{dist}(\texttt{V},\texttt{D1})$ and $\texttt{dist}(\texttt{V},\texttt{D2})$ are positive user-defined antecedents. The negative ground atom $\texttt{del}(\texttt{dist}(\texttt{a},5))$ is an example of a negative assertion. □

Operational Semantics

A Logical Algorithms execution state σ consists of a set of (positive and negative) assertions. A state can simultaneously contain the positive assertion A and the negative assertion $\texttt{del}(A)$. Let $\mathcal{D}$ be the usual interpretation for the comparisons. Table 5.1 shows the (single) transition of the Logical Algorithms operational semantics for a given program P.

A state is called final if no more transitions apply to it. A non-final state has priority p if the next firing rule instance has priority p. The condition $\theta(C) \not\subseteq \sigma$ ensures that no rule instance fires more than once and prevents trivial non-termination. This condition, combined with the fact that each transition only creates new assertions, causes the consecutive states in a derivation to be monotone increasing. Although the priorities restrict the possible derivations, the choice of which rule instance to fire from those with equal priority is non-deterministic.

Differences compared to CHR$^{\text{rp}}$

Logical Algorithms differs from CHR$^{\text{rp}}$ in the following ways:

- A Logical Algorithms state is a set of ground assertions, while the CHR constraint store is a multi-set and may also contain non-ground constraints.
- In Logical Algorithms, built-in constraints are restricted to ask constraints and only include comparisons; CHR^{rp} supports any kind of built-in constraints.
- A removed CHR constraint may be reasserted and can then participate again in rule firings whereas a removed LA assertion cannot be asserted again.
- A Logical Algorithms rule may contain negated heads. In contrast, CHR^{rp} requires all heads to be positive.[1]
- In the Logical Algorithms language, the priority of a dynamic priority rule is determined by the variables in the left-most head, whereas in CHR^{rp} it may depend on multiple heads.

We note that rules for which the priority depends on more than one head, can easily be transformed into the correct form as follows. Given a Logical Algorithms rule of the form

$$r \ @ \ p : A_1, \ldots, A_m, A_{m+1}, \ldots, A_n \Rightarrow C$$

where the priority expression p is fully determined by the variables from the antecedents $A_1, \ldots, A_m$. This rule can be transformed into the equivalent rules

$$r_1 \ @ \ 1 : A_1, \ldots, A_m \Rightarrow \texttt{priority}(r, p)$$
$$r_2 \ @ \ p : \texttt{priority}(r, p), A_1, \ldots, A_m, A_{m+1}, \ldots, A_n \Rightarrow C$$

where `priority`/2 is a new user-defined predicate. Now the first head of the dynamic priority rule determines the rule priority. Note that this transformation does not increase overall complexity: it only results in the first m heads to be matched with twice.

5.2.2 Meta-Complexity Results for LA and CHR^{rp}

The Logical Algorithms language was designed with a meta-complexity result in mind. Such a result has also been formulated for CHR. In this subsection, we review both results and give a first intuition on how they relate to each other.

The Logical Algorithms Meta-Complexity Result

A *prefix instance* of a Logical Algorithms rule $r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$ is a tuple $\langle \theta(r), i \rangle$ with θ a ground substitution defined on the variables occurring in $A_1, \ldots, A_i$ and $1 \leq i \leq n$. Its antecedents are $\theta(A_1), \ldots, \theta(A_i)$. A *strong prefix*

[1] See (Van Weert et al. 2006) for an extension of CHR with negation as absence.

firing is a prefix instance whose antecedents hold in a state with priority lower or equal to the prefix' rule priority.[2] The time complexity for running Logical Algorithms programs is given by Ganzinger and McAllester (2002) as $\mathcal{O}(|\sigma_0| + P_\mathsf{s} + (P_\mathsf{d} + A_\mathsf{d}) \cdot \log N)$ where σ_0 is the initial state and $|\sigma_0|$ is its size. P_s is the number of strong prefix firings of static priority rules and P_d is the number of strong prefix firings of dynamic priority rules; A_d is the number of assertions that may participate in a dynamic priority rule instance; and N is the number of distinct priorities. The following example is adapted from (Ganzinger and McAllester 2002).

Example 5.2 (Dijkstra's Shortest Path) Listing 5.1 shows an implementation of Dijkstra's single-source shortest path algorithm in LA.

```
d1 @ 1   : source(V) => dist(V,0).
d2 @ 1   : dist(V,D1), dist(V,D2), D2 < D1 => del(dist(V,D1)).
d3 @ D+2 : dist(V,D), e(V,C,U) => dist(U,D+C).
```

Listing 5.1: Dijkstra's shortest path algorithm in Logical Algorithms

The code is very similar to the CHR^{rp} code of Listing 3.2. A `source`(v) fact means that v is the (unique) source node for the algorithm. A `dist`(v, d) fact means that the shortest path distance from the source node to node v does not exceed d. Finally, an `e`(v, c, u) fact means there is an edge from node v to node u with cost (weight) c. Given an initial state consisting of one `source`/1 fact and e `e`/3 facts, we can derive that the number of strong prefix firings is $\mathcal{O}(1)$ for rule `d1`, and $\mathcal{O}(e)$ for both rules `d2` and `d3`. This result is based on the fact that at priority 2 and lower, there is at most one (positive) `dist`/2 fact for each node, and each of these facts represent the shortest path distance from the source node to this node. This means that at most e `dist`/2 facts are ever created. Using the meta-complexity theorem, we find that the total complexity is $\mathcal{O}(e \log e)$. □

The "As Time Goes By" Approach

In (Frühwirth 2002a; Frühwirth 2002b), an upper-bound on the worst case time complexity of a CHR program P is given as

$$\mathcal{O}\left(D \sum_{r \in P} (c_{\mathsf{max}}^{n_r} (O_{H_r} + O_{G_r}) + (O_{C_r} + O_{B_r}))\right) \tag{5.1}$$

where D is the maximal derivation length (i.e., the maximal number of rule firings), c_{max} is the maximal number of CHR constraints in the store, and for each rule $r \in P$:

[2] In (Ganzinger and McAllester 2002), also the concept of a *weak* prefix firing is defined, but it is of no importance for our purposes.

- n_r is the number of heads in r
- O_{H_r} is the cost of head matching, i.e. checking that a given sequence of n_r constraints match with the n_r heads of rule r
- O_{G_r} is the cost of checking the guard
- O_{C_r} is the cost of adding built-in constraints after firing
- O_{B_r} is the cost of adding and removing CHR constraints after firing

For programs with simplification and simpagation rules only, the maximal derivation length can be derived using an appropriate ranking on constraints that decreases after each rule firing (Frühwirth 2000). We note that finding such a ranking is not trivial. The meta-complexity result is based on a very naive CHR implementation, and therefore on the one hand gives an upper-bound on the time complexity for any reasonable implementation of CHR, but on the other hand often largely overestimates the worst case time complexity on optimized implementations.[3] The following example is adapted from (Frühwirth 2002a).

Example 5.3 (Boolean) The rules below implement the Boolean `and`$(x, y, x \wedge y)$ constraint given that 1 represents *true* and 0 represents *false*.

```
and(0,Y,Z) <=> Z = 0.          and(X,0,Z) <=> Z = 0.
and(X,1,Z) <=> X = Z.          and(1,Y,Z) <=> Y = Z.
and(X,X,Z) <=> X = Z.          and(X,Y,1) <=> X = 1, Y = 1.
```

Let the rank of an `and`/3 constraint be one, then the rank of the head of each rule equals one, and the rank of the body equals zero.[4] For a goal consisting of n `and`/3 constraints, the derivation length is n, which is also the maximal number of CHR constraints in the store. The cost of head matching, (implicit) guard checking, removing CHR constraints and asserting built-in constraints can all be considered constant. Then using (5.1), we derive that the total runtime complexity is $\mathcal{O}(n^2)$. □

A First Comparison

Although at this point we do not intend to make a complete comparison between both results, we can already show that the Logical Algorithms result in a sense is at least as accurate as Frühwirth's approach, at least for programs without built-in tell constraints. The reasoning is as follows. In each derivation step, a constant

[3] Built-in constraints may lead to a worse complexity in practical optimized implementations if many constraints are repeatedly reactivated without this resulting in new rule firings. We return to this issue in Section 5.6.3.

[4] Built-in constraints have a rank of zero by definition.

number of atoms (constraints) are asserted. Let c_{max} be the maximal number of (strictly) positive atoms in the database in any given state. Furthermore assume rules have positive heads only, then each of the asserted atoms can participate in at most $\sum_{r \in P} \left(n_r \cdot c_{\mathsf{max}}^{n_r - 1} \right)$ strong prefix firings. Because only $\mathcal{O}(c+D)$ constraints are ever asserted where c is the number of CHR constraints in the initial goal and D is the derivation length, the total number of strong prefix firings $P_s + P_d$ is

$$\mathcal{O}\left((c + D) \cdot \sum_{r \in P} c_{\mathsf{max}}^{n_r - 1} \right)$$

and because $c = \mathcal{O}(c_{\mathsf{max}})$ we also have the following bound

$$\mathcal{O}\left(D \cdot \sum_{r \in P} c_{\mathsf{max}}^{n_r} \right) \tag{5.2}$$

In the absence of (dynamic) priorities, the total runtime complexity according to the Logical Algorithms meta-complexity result is bounded by the same formula (5.2) and hence is at least as accurate as the result of Frühwirth (2002b) given that the cost of both head matching (O_{H_r}) and adding and removing CHR constraints (O_{B_r}) is constant for each rule r.

5.3 Translating Logical Algorithms into CHR$^{\text{rp}}$

In this section, we show how Logical Algorithms programs can be translated into CHR$^{\text{rp}}$ programs. CHR states of the translated program can be mapped onto LA states of the original. With respect to this mapping, both programs have the same derivations.

5.3.1 Translation Schema

The translation of a LA program P is denoted by $T(P) = T_{S+D}(P) \cup T_R(P)$. The definitions of $T_{S+D}(P)$ and $T_R(P)$ are given below.

Set and Deletion Semantics

We represent Logical Algorithms assertions as CHR constraints consisting of the assertion itself and an extra argument, called the *mode indicator*, denoting whether it is positively asserted (‘`p`’), negatively asserted (‘`n`’) or both (‘`b`’). For every user-defined predicate a/n occurring in P, $T_{S+D}(P)$ contains the following rules to deal

with a new positive or negative assertion:

$$\begin{aligned} 1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus a(\bar{X}) &\iff M \neq \mathtt{n} \mid \mathtt{true} \\ 1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{n}), a(\bar{X}) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \\ 2 :: a(\bar{X}) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{p}) \end{aligned}$$

$$\begin{aligned} 1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus \mathtt{del}(a(\bar{X})) &\iff M \neq \mathtt{p} \mid \mathtt{true} \\ 1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{p}), \mathtt{del}(a(\bar{X})) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \\ 2 :: \mathtt{del}(a(\bar{X})) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{n}) \end{aligned}$$

If a representation already exists, one of the priority 1 rules updates this representation. Otherwise, one of the priority 2 rules generates a new representation. At lower priorities, it is guaranteed that every assertion, whether asserted positively, negatively or both, is represented by exactly one constraint in the store.

Rules

Given a LA rule $r \in P$ of the form

$$r \mathbin{@} p : A_1, \ldots, A_n \Rightarrow C$$

We first split up the antecedents into user-defined antecedents and comparison antecedents by using the split function defined below.

$$\begin{aligned} \mathsf{split}([A|T]) = &\begin{cases} \langle [A|A^u], A^c \rangle & \text{if } A \text{ is a user-defined atom} \\ \langle A^u, [A|A^c] \rangle & \text{if } A \text{ is a comparison} \end{cases} \\ &\text{where } \mathsf{split}(T) = \langle A^u, A^c \rangle \\ \mathsf{split}(\epsilon) = &\langle \epsilon, \epsilon \rangle \end{aligned}$$

In the Logical Algorithms language, a given assertion may participate multiple times in the same rule instance, whereas in CHR all constraints in a single rule instance must be different. To overcome this semantic difference, a single LA rule is translated as a set of CHR rules such that every CHR rule covers a case of syntactically equal head constraints. Let $\langle A^u, A^c \rangle = \mathsf{split}([A_1, \ldots, A_n])$ with $A^u = [A^u_1, \ldots, A^u_m]$ and $A^c = [A^c_1, \ldots, A^c_l]$. Let $\mathcal{P}$ be the set of all partitions of $\{1, \ldots, m\}$.[5] For a given partition $\rho \in \mathcal{P}$, the following function returns the most general unifier that unifies all antecedents $\{A_i \mid i \in S\}$ for every $S \in \rho$ where $\mathsf{mgu}(S)$ is the most general unifier of all elements in S.

$$\mathsf{partition\_to\_mgu}(\rho, [A^u_1, \ldots, A^u_m]) = \bigcirc_{S \in \rho} \mathsf{mgu}(\{A^u_i \mid i \in S\})$$

[5] $\mathcal{P}$ contains B_m elements in the worst case with B_m the m^{th} Bell number.

Let $\mathcal{PU} = \{\langle \rho, \theta\rangle \mid \rho \in \mathcal{P} \wedge \theta = \mathsf{partition\_to\_mgu}(\rho, A^u) \wedge \mathcal{D} \models \bar{\exists}_{\emptyset}\theta(A^c)\}$. $\mathcal{PU}$ contains all partitions for which $\mathsf{partition\_to\_mgu}$ is defined and for which the comparison antecedents A^c are still satisfiable after applying the unifier. The next step is to filter out antecedents so that every set in the partition has only one representative. This is done by computing $\mathsf{filter}(\theta(A^u), \rho)$ for each $\langle \rho, \theta\rangle \in \mathcal{PU}$ where the filter function is as follows:

$$\mathsf{filter}([\theta(A_i^u)|T], \rho) = \begin{cases} [\theta(A_i^u)|\mathsf{filter}(T, \rho)] & \text{if } \exists S \in \rho : i = \min(S) \\ \mathsf{filter}(T, \rho) & \text{otherwise} \end{cases}$$
$$\mathsf{filter}(\epsilon, \_) = \epsilon$$

Finally, we add mode indicators to all remaining user-defined antecedents:

$$\mathsf{modes}([A^{u'}|T]) = \begin{cases} \langle [a_{\mathtt{r}}(\bar{X}, \mathtt{p})|A^m], N\rangle & \text{if } A^{u'} = a(\bar{X}) \\ \langle [a_{\mathtt{r}}(\bar{X}, N')|A^m], [N' \neq \mathtt{p}|N]\rangle & \text{if } A^{u'} = \mathtt{del}(a(\bar{X})) \end{cases}$$
$$\text{where } \langle A^m, N\rangle = \mathsf{modes}(T)$$
$$\mathsf{modes}(\epsilon) = \langle \epsilon, \epsilon\rangle$$

The modes function returns both the resulting antecedents and the necessary conditions on the mode indicators of these antecedents. For every $\langle \rho, \theta\rangle \in \mathcal{PU}$, the CHR translation $T_R(P)$ contains a rule

$$p + 2 :: r_\rho \ @\ H \implies g_1, g_2 \mid C'$$

where $\langle H, g_1\rangle = \mathsf{modes}(\mathsf{filter}(\theta(A^u), \rho))$, $g_2 = \theta(A^c)$ and $C' = \theta(C)$.

Examples

We illustrate the translation schema on some examples.

Example 5.4 The translation of the LA implementation of Dijkstra's shortest path algorithm given in Listing 5.1 is given in Listing 5.2. □

Example 5.5 The following rule is part of the union-find implementation given in (Ganzinger and McAllester 2002).

```
uf4 @ 1 : union(X,Y), find(X,Z), find(Y,Z) => del(union(X,Y)).
```

Because antecedents $\mathtt{find}(\mathtt{X}, \mathtt{Z})$ and $\mathtt{find}(\mathtt{Y}, \mathtt{Z})$ are unifiable, their translation to CHR$^{\mathrm{rp}}$ is as follows:

```
3 :: uf4_{1/2/3} @ union_r(X,Y,p), find_r(X,Z,p), find_r(Y,Z,p) ==> del(union(X,Y)).
3 :: uf4_{1/23}  @ union_r(X,X,p), find_r(X,Z,p) ==> del(union(X,X)).
```

□

```
1 :: e_r(V,C,U,M) \ e(V,C,U) <=> M \= n | true.
1 :: e_r(V,C,U,n) , e(V,C,U) <=> e_r(V,C,U,b).
2 ::                e(V,C,U) <=> e_r(V,C,U,p).

1 :: e_r(V,C,U,M) \ del(e(V,C,U)) <=> M \= p | true.
1 :: e_r(V,C,U,p) , del(e(V,C,U)) <=> e_r(V,C,U,b).
2 ::                del(e(V,C,U)) <=> e_r(V,C,U,n).

... % (similar rules for source/1 and dist/2)

  3 :: d1_1   @ source_r(V,p) ==> dist(V,0).
  3 :: d2_1/2 @ dist_r(V,D_1,p), dist_r(V,D_2,p) ==> D_2 < D_1 | del(dist(V,D_1)).
D+4 :: d3_1/2 @ dist_r(V,D,p), e_r(V,C,U,p) ==> dist(U,D+C).
```

Listing 5.2: Translation to CHR^{rp} of the program of Listing 5.1

5.3.2 Correspondence between LA and ω_p Derivations

In this subsection, we show that every derivation of the original program under the Logical Algorithms semantics, corresponds to a derivation of the translation under the ω_p semantics of CHR^{rp}. In order to do so, we introduce a mapping function chr_to_la between *reachable* CHR execution states and Logical Algorithms states.[6] Reachability is considered with respect to initial states of the form $\langle G, \emptyset, \texttt{true}, \emptyset\rangle_n$ where the user-defined constraints in G are of the form $a(\bar{X})$ and $\texttt{del}(a(\bar{X}))$ and do not include constraints of the form $a_{\texttt{r}}(\bar{X}, M)$.

$$\begin{aligned}\mathsf{chr\_to\_la}(\sigma) = \{&a(\bar{X}) \mid a(\bar{X}) \in A \vee (a_{\texttt{r}}(\bar{X}, M) \in A \wedge M \neq \texttt{n})\}\\ &\cup \{\texttt{del}(a(\bar{X})) \mid \texttt{del}(a(\bar{X})) \in A \vee (a_{\texttt{r}}(\bar{X}, M) \in A \wedge M \neq \texttt{p})\}\end{aligned}$$

where $\sigma = \langle G, S, B, T\rangle_n$ and $A = G \cup \mathsf{chr}(S)$. The mapping function also takes into account the constraints that are still in the goal and those for which the set and deletion semantics rules have not yet fired. In the rest of this section, we first show how CHR execution states are normalized and then show that in a Logical Algorithms state and its corresponding normalized CHR execution state, corresponding rule instances can fire. We start by defining a *pre-normal form*.

Definition 4 (Pre-normal Form) *A (reachable) state σ is in pre-normal form if and only if $\sigma = \langle \emptyset, S, \texttt{true}, T\rangle_n$, all constraints in S are of the form $a_r(\bar{X}, M)\#i$, and if $a_r(\bar{X}, M_1)\#i_1 \in S$ and $a_r(\bar{X}, M_2)\#i_2 \in S$ then $i_1 = i_2$ (and consequently $M_1 = M_2$).*

The following lemma shows that every reachable state is *pre-normalized* before rules are tried with priority > 2.

[6] See (Duck et al. 2007) for a formal definition of reachability.

Lemma 1 (Pre-normalization) *For every reachable state σ, there exists a finite derivation $D = \sigma \overset{\omega_p}{\rightarrowtail}^*_{T(P)} \sigma^*$ such that σ^* is in pre-normal form, $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma^*)$, and all rules fired in D have priority 1 or 2. Every state has a unique pre-normal form with respect to the $\mathsf{chr\_to\_la}$ mapping function.*

Proof: We introduce the following ranking function on CHR states:

$$\|\sigma\| = 2 \cdot \left|\{a(\bar{X}) \mid a(\bar{X}) \in A\} \uplus \{\mathtt{del}(a(\bar{X})) \mid \mathtt{del}(a(\bar{X})) \in A\}\right| + |G|$$

where $\sigma = \langle G, S, \mathtt{true}, T\rangle_n$, $A = G \uplus \mathsf{chr}(S)$ and if X is a (multi-)set, $|X|$ is its cardinality. Clearly, the rank of any state is positive, and if $\|\sigma\| = 0$, state σ is in pre-normal form. If σ is not in pre-normal form, then there exists at least one transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$. We show that for all such transitions $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ and $\|\sigma'\| < \|\sigma\|$, which ensures termination.

If the goal G is not empty, then only the **Introduce** transition is applicable. Every application of this transition moves a CHR constraint from the goal to the CHR constraint store, so $\|\sigma'\| = \|\sigma\| - 1$. By definition, $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$ (because the $\mathsf{chr\_to\_la}$ function does not distinguish between the goal and the CHR constraint store).

If the goal G is empty then given that σ is not in pre-normal form, $\mathsf{chr}(S)$ contains a constraint of the form $a(\bar{X})$ or $\mathtt{del}(a(\bar{X}))$. We look into detail to the case of $a(\bar{X}) \in \mathsf{chr}(S)$; the case of $\mathtt{del}(a(\bar{X})) \in \mathsf{chr}(S)$ is similar. We start by showing that at least one rule of priority 1 or 2 is applicable. Next, we show that each rule application decreases the norm and maintains the invariance with respect to the $\mathsf{chr\_to\_la}$ function.

Assume $a(\bar{X}) \in \mathsf{chr}(S)$. If $a_{\mathtt{r}}(\bar{X}, \mathtt{p}) \in \mathsf{chr}(S)$ or $a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \in \mathsf{chr}(S)$ then the following rule of $T(P)$ is applicable:

$$1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus a(\bar{X}) \iff M \neq \mathtt{n} \mid \mathtt{true}$$

If $a_{\mathtt{r}}(\bar{X}, \mathtt{n}) \in \mathsf{chr}(S)$ then the rule below applies:

$$1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{n}), a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{b})$$

Finally, if no rule of priority 1 can be applied, which implies that no constraint of the form $a_{\mathtt{r}}(\bar{X}, M) \in \mathsf{chr}(S)$, then the following $T(P)$ rule can fire:

$$2 :: a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{p})$$

This covers all possibilities. Now we look at what happens after firing one of the priority 1 or 2 rules. The rule

$$1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus a(\bar{X}) \iff M \neq \mathtt{n} \mid \mathtt{true}$$

removes a constraint $a(\bar{X})\#i$ from S and has an empty body, so $\|\sigma'\| = \|\sigma\| - 2$. Since $M \neq \mathtt{n}$ the removed constraint was already represented by the $a_{\mathtt{r}}(\bar{X}, M)$ constraint and so $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$. Firing

$$1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{n}), a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{b})$$

causes the removal of two constraints from S, namely $a_{\mathtt{r}}(\bar{X}, \mathtt{n})\#i$ and $a(\bar{X})\#j$. Furthermore, it adds a new constraint $a_{\mathtt{r}}(\bar{X}, \mathtt{b})$ to G. This results in $\|\sigma'\| = \|\sigma\| - 1$. The new constraint represents the combined mode of both removed constraints and hence $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$. Finally, the rule

$$2 :: a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{p})$$

is only applicable if $\mathsf{chr}(S)$ does not contain a constraint of the form $a_{\mathtt{r}}(\bar{X}, M)$. It removes a constraint $a(\bar{X})\#i$ from S and adds a new constraint $a_{\mathtt{r}}(\bar{X}, \mathtt{p})$ to G, resulting in $\|\sigma'\| = \|\sigma\| - 1$. The new representation covers the positive assertion and so $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$.

In summary, if the goal is empty and σ is not in pre-normal form, a rule of priority 1 or 2 can fire and so no rule with lower priority is applicable. All applicable transitions strictly decrease the value of the ranking function and so the pre-normalization terminates. Finally, none of the possible transitions changes the value of $\mathsf{chr\_to\_la}$. □

The state σ^* is called a pre-normalization of σ.

Definition 5 (Implied Rule Instance) *A rule instance $\theta(r)$ is implied in a state σ if $\theta(C) \subseteq \mathsf{chr\_to\_la}(\sigma)$ with $\theta(C)$ the conclusion of $\theta(r)$.*

Lemma 2 (Normalization) *Let there be given a pre-normalized state $\sigma = \langle \emptyset, S, \mathtt{true}, T\rangle_n$. If there exists a transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$ in which an implied rule instance fires, then the pre-normalization of σ' has the form $\langle \emptyset, S, \mathtt{true}, T'\rangle_{n'}$ with $T' \supsetneq T$. In other words $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ and the CHR constraint store after pre-normalization is unchanged from the one before the implied rule instance fired while the propagation history is increased.*

Proof: Let $\theta(r)$ be the implied rule instance with conclusion $\theta(C)$. Since $\theta(C) \subseteq \mathsf{chr\_to\_la}(\sigma)$ with $\sigma = \langle \emptyset, S, \mathtt{true}, T\rangle_n$, we have $\sigma' = \langle \theta(C), S, \mathtt{true}, T \cup \{t\}\rangle_n$ and $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ with t the propagation history tuple corresponding to $\theta(r)$. The goal G of σ' equals $\theta(C)$ and so it holds that if $a(\bar{X}) \in G$ then $a_{\mathtt{r}}(\bar{X}, \mathtt{p}) \in \mathsf{chr}(S)$ or $a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \in \mathsf{chr}(S)$ and if $\mathtt{del}(a(\bar{X})) \in G$ then $a_{\mathtt{r}}(\bar{X}, \mathtt{n}) \in \mathsf{chr}(S)$ or $a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \in \mathsf{chr}(S)$. Now all constraints in the goal are first introduced in the CHR constraint store. Next, the newly introduced CHR constraints are removed one by one using one of the following rules:

$$1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus a(\bar{X}) \iff M \neq \mathtt{n} \mid \mathtt{true}$$
$$1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus \mathtt{del}(a(\bar{X})) \iff M \neq \mathtt{p} \mid \mathtt{true}$$

These rules remove all the constraints that were introduced from the goal and do not change the rest of the CHR constraint store, hence after pre-normalization, the CHR constraint store equals that of state σ again. □

Because the CHR constraint store remains unchanged after firing an implied rule instance and pre-normalizing the resulting state, only finitely many such rule instances can fire before either reaching a final execution state, or a state in which a non-implied rule instance can fire. We call such a state *normalized*.

Definition 6 (Normal Form) *A pre-normalized CHR execution state σ is in normal form if it is a final state ($\sigma \overset{\omega_p}{\not\rightarrowtail}_{T(P)}$) or there exists a transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$ such that $\mathsf{chr\_to\_la}(\sigma') \not\supseteq \mathsf{chr\_to\_la}(\sigma)$, i.e., in which a non-implied rule instance is fired.*

Lemma 3 *For every Logical Algorithms state σ_{LA} and every normalized CHR execution state $\sigma = \langle \emptyset, S, \mathtt{true}, T \rangle_n$ such that $\sigma_{LA} = \mathsf{chr\_to\_la}(\sigma)$, there exists a transition $\sigma_{LA} \overset{LA}{\rightarrowtail}_P \sigma'_{LA}$ if and only if there exists a transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$ firing a non-implied rule instance such that $\sigma'_{LA} = \mathsf{chr\_to\_la}(\sigma')$.*

Proof: A transition of σ_{LA} to σ'_{LA} implies there exists an applicable rule instance $\theta(r)$ of a rule r in P with priority p of the form

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

Let $\langle A^u, A^c \rangle = \langle [A^u_1, \ldots, A^u_m], [A^c_1, \ldots, A^c_l] \rangle = \mathsf{split}([A_1, \ldots, A_n])$ where we use the split function defined in Section 5.3.1. The user-defined antecedents can be partitioned into sets of syntactically equal antecedents with respect to the matching substitution θ. The following function returns this partition:

$$\mathsf{substitution\_to\_partition}(\theta, [A^u_1, \ldots, A^u_m]) = \{S_1, \ldots, S_m\}$$

where $S_i = \{j \mid \theta(A^u_i) = \theta(A^u_j)\}$. Let $\rho = \mathsf{substitution\_to\_partition}(\theta, A^u)$. From the partition, we find the most general unifier θ' that unifies all antecedents $\{A^u_i \mid i \in S\}$ for every $S \in \rho$: $\theta' = \mathsf{partition\_to\_mgu}(\rho, A^u)$ with $\mathsf{partition\_to\_mgu}$ as defined in Section 5.3.1. Clearly, θ' exists and is more general than θ. The applicability of the **Apply** transition means that for all comparison antecedents A^c_i with $1 \leq i \leq l$, $\mathcal{D} \models \theta(A^c_i)$ and so it holds that $\mathcal{D} \models \bar{\exists}_{\emptyset} \theta'(A^c_1 \wedge \ldots \wedge A^c_l)$ and consequently a rule r_ρ exists. This rule looks as follows:

$$p + 2 :: r_\rho \ @ \ H_1, \ldots, H_k \implies g_1, g_2 \mid C'$$

with $\langle [H_1, \ldots, H_k], g_1 \rangle = \mathsf{modes}(A^f)$, $A^f = [A^f_1, \ldots, A^f_k] = \mathsf{filter}(\theta'(A^u), \rho)$, $g_2 = \theta'(A^c)$ and $C' = \theta'(C)$. The modes and filter functions are as defined in Section 5.3.1.

Let θ'' be a ground matching substitution such that $\theta = \theta''|_{\mathsf{vars}(\theta)} \circ \theta'$ where $\theta''|_{\mathsf{vars}(\theta)}$ is the projection of θ'' on the variables in θ. Since θ' is more general than θ, θ'' exists. For all $i \in \{1, \ldots, k\}$, if $A_i^f = a(\bar{X})$ then $H_i = a_{\mathtt{r}}(\bar{X}, \mathtt{p})$. Because of the applicability of Logical Algorithms rule r in state σ_{LA}, $\theta''(a(\bar{X})) \in \sigma_{\mathrm{LA}}$ and $\theta''(\mathtt{del}(a(\bar{X}))) \notin \sigma_{\mathrm{LA}}$, so $H_i' = \theta''(a_{\mathtt{r}}(\bar{X}, \mathtt{p}))\#id_i \in S$ and $\theta''(H_i) = \mathsf{chr}(H_i')$. Similarly, if $A_i^f = \mathtt{del}(a(\bar{X}))$ then $H_i = a_{\mathtt{r}}(\bar{X}, N)$ and g_1 contains $N \neq \mathtt{p}$; $\theta''(\mathtt{del}(a(\bar{X}))) \in \sigma_{\mathrm{LA}}$ and as a result $H_i' = \theta''(a_{\mathtt{r}}(\bar{X}, N'))\#id_i \in S$ with $N' = \mathtt{n}$ or $N' = \mathtt{b}$. Since N only appears in H_i and the guard $N \neq \mathtt{p}$, we can further impose that $\theta''(N) = N'$ and then $\theta''(H_i) = \mathsf{chr}(H_i')$.

All $\theta''(A_i^f)$ are different for $1 \leq i \leq k$, and therefore, all id_i must be different. From $\mathcal{D} \models \bar{\exists}_{\emptyset}\theta(A_i^c)$ for $1 \leq i \leq l$ and because $\theta''(g_1) = [N_1 \neq \mathtt{p}, \ldots, N_o \neq \mathtt{p}]$ with $N_j = \mathtt{n}$ or $N_j = \mathtt{b}$ for $1 \leq j \leq o$, $\mathcal{D} \models \mathtt{true} \rightarrow \bar{\exists}_{\emptyset}\theta''(g_1 \wedge g_2)$. We conclude that θ'' is a ground matching substitution that matches the head with constraints from S and for which the guard is entailed.

It is not possible that $\langle \mathsf{id}(H), \epsilon, r_\rho \rangle \in T$ because $\mathsf{chr\_to\_la}$ grows monotonically, which implies that $\theta(C) = \theta''(C') \in \mathsf{chr\_to\_la}(\sigma) = \sigma_{\mathrm{LA}}$ which contradicts with the applicability of $\theta(r)$ in σ_{LA}.

If we ignore rule priorities, all conditions are satisfied so that rule instance $\theta(r_\rho)$ can fire. The resulting state σ' has the form $\langle \theta(C), S, \mathtt{true}, T \cup \{\langle \mathsf{id}(H), \epsilon, r_\rho \rangle\} \rangle_n$. Clearly, if $\sigma_{\mathrm{LA}} = \mathsf{chr\_to\_la}(\langle \emptyset, S, \mathtt{true}, T \rangle_n)$ and $\sigma'_{\mathrm{LA}} = \sigma_{\mathrm{LA}} \cup \theta(C)$ then $\sigma'_{\mathrm{LA}} = \mathsf{chr\_to\_la}(\sigma')$. We now prove that every CHR transition firing a non-implied rule instance corresponds to a Logical Algorithms transition, also ignoring rule priorities. Both results combined give us that the priority of the highest priority rule instance is equal in both σ and σ_{LA}.

A transition of $\sigma = \langle \emptyset, S, \mathtt{true}, T \rangle_n$ to σ' implies that $T(P)$ contains a rule

$$p + 2 :: r_\rho \ @ \ H \implies g_1, g_2 \mid C'$$

and so the Logical Algorithms program P contains a rule

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

Let $\langle A^u, A^c \rangle = \mathsf{split}([A_1, \ldots, A_n])$ and $\theta = \mathsf{partition\_to\_mgu}(\rho, A^u)$. If $A_i = a(\bar{X}) \in A^u$ then $\theta(a_{\mathtt{r}}(\bar{X}, \mathtt{p})) \in H$. If $A_i = \mathtt{del}(a(\bar{X})) \in A^u$ then $\theta(a_{\mathtt{r}}(\bar{X}, N)) \in H$ and $(N \neq \mathtt{p}) \in g_1$. Finally, if $A_i \in A^c$ then $\theta(A_i) \in g_2$. There exists a (ground) matching substitution θ' such that $\theta'(H) \in \mathsf{chr}(S)$ and $\mathcal{D} \models \bar{\exists}_{\emptyset}\theta'(g_1 \wedge g_2)$.

Let $\theta'' = \theta' \circ \theta$ and let $\sigma_{\mathrm{LA}} = \mathsf{chr\_to\_la}(\sigma)$. Because θ' is a ground substitution, $\mathcal{D} \models \bar{\exists}_{\emptyset}\theta'(g_1 \wedge g_2)$ implies that for all $A_i \in A^c$, $\mathcal{D} \models \theta''(A_i)$. For all positive user-defined antecedents $A_i = a(\bar{X}) \in A^u$, we have that $\theta''(a_{\mathtt{r}}(\bar{X}, \mathtt{p})) \in \mathsf{chr}(S)$ and so $\theta''(A_i) \in \sigma_{\mathrm{LA}}$ and $\mathtt{del}(\theta''(A_i)) \notin \sigma_{\mathrm{LA}}$. For all negative user-defined antecedents $A_i = \mathtt{del}(a(\bar{X})) \in A^u$, we have that $\theta''(a_{\mathtt{r}}(\bar{X}, N)) \in \mathsf{chr}(S)$ with $N = \mathtt{b}$ or $N = \mathtt{n}$ and so $\theta''(A_i) \in \sigma_{\mathrm{LA}}$. We have assumed that $\theta'(r_\rho)$ is not an implied rule instance and so $\theta'(C') = \theta''(C) \not\subseteq \sigma_{\mathrm{LA}}$.

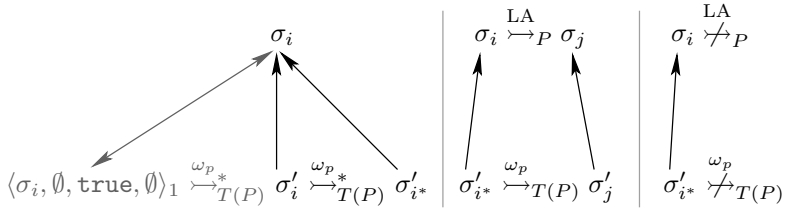


Figure 5.1: Correspondence between derivations in Logical Algorithms and CHR$^{\mathrm{rp}}$

If we again ignore rule priorities, all conditions are satisfied so that rule instance $\theta''(r)$ can fire in state σ_{LA} and it holds that $\sigma'_{\mathrm{LA}} = \sigma_{\mathrm{LA}} \cup \theta''(C) = \mathsf{chr\_to\_la}(\sigma')$ since $\sigma' = \langle \theta'(C'), S, \mathtt{true}, T \cup \{\langle \mathsf{id}(H), \epsilon, r_\rho \rangle\} \rangle_n$. Now we have that both the original program P and its translation $T(P)$ can fire corresponding rule instances if we ignore priorities, and so their highest priority rule instances also correspond. □

Theorem 7 *For every reachable CHR^{rp} state σ, if $\sigma \overset{\omega_p}{\longmapsto}_{T(P)} \sigma'$ then it holds that either $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ or $\mathsf{chr\_to\_la}(\sigma) \overset{LA}{\longmapsto}_P \mathsf{chr\_to\_la}(\sigma')$.*

Proof: Implied by Lemmas 1, 2 and 3. □

Theorem 8 *For every Logical Algorithms state σ_i and reachable CHR^{rp} state σ'_i such that $\mathsf{chr\_to\_la}(\sigma'_i) = \sigma_i$, there exists a finite CHR^{rp} derivation $\sigma'_i \overset{\omega_p}{\longmapsto}^*_{T(P)} \sigma'_{i*}$ for which holds that $\mathsf{chr\_to\_la}(\sigma'_{i*}) = \sigma_i$ such that if $\sigma_i \overset{LA}{\longmapsto}_P \sigma_j$ then $\sigma'_{i*} \overset{\omega_p}{\longmapsto}_{T(P)} \sigma'_j$ with $\mathsf{chr\_to\_la}(\sigma'_j) = \sigma_j$ and if σ_i is a final state then σ'_{i*} is also a final state.*

Proof: Implied by Lemmas 1, 2 and 3. □

Given a Logical Algorithms state σ, we can use $\langle \sigma, \emptyset, \mathtt{true}, \emptyset \rangle_1$ as initial state for the CHR$^{\mathrm{rp}}$ derivation. Theorem 8 is illustrated by Figure 5.1.

5.3.3 Relation with Weak Bisimilarity

To capture the meaning of the above correspondence results, we relate them to the notion of (weak) bisimulation. A bisimulation is a relation between the states of a labeled transition system (LTS). A relation $R \subseteq S_1 \times S_2$ between the states in S_1 and those in S_2 is a bisimulation if $p \; R \; q$ and $p \overset{\alpha}{\rightarrow} p'$ implies that $q \overset{\alpha}{\rightarrow} q'$ with $p' \; R \; q'$, and similarly, $p \; R \; q$ and $q \overset{\alpha}{\rightarrow} q'$ implies that $p \overset{\alpha}{\rightarrow} p'$ with $p' \; R \; q'$. Here, α is the label of the transition $p \overset{\alpha}{\rightarrow} p'$ from state p to state p'. If a transition from p to p' has no observable effect, it is called a *silent* transition and denoted by $p \overset{\tau}{\rightarrow} p'$. A relation $R \subseteq S_1 \times S_2$ is a *weak* bisimulation if $p \; R \; q$ and $p \overset{\alpha}{\rightarrow} p'$ implies that $q \overset{\tau}{\rightarrow}^* q_* \overset{\alpha}{\rightarrow} q'_* \overset{\tau}{\rightarrow}^* q'$ with $p' \; R \; q'$, and vice versa with the roles of p

and q swapped. Here $p \xrightarrow{\tau}{}^{*} p'$ means p and p' are linked by zero or more silent transitions.

Let S_1 be the set of valid Logical Algorithms states for program P and let $S_2 = \{\mathsf{chr\_to\_la}(\sigma) \mid \langle G, \emptyset, \mathtt{true}, \emptyset\rangle_1 \overset{\omega_p}{\rightarrowtail}{}^{*}_{T(P)} \sigma \wedge G \in S_1\}$, i.e., S_2 is found by applying the $\mathsf{chr\_to\_la}$ mapping function to all reachable CHR$^{\text{rp}}$ states for program $T(P)$. We transform the state transition systems for Logical Algorithms and CHR$^{\text{rp}}$ to labeled transition systems as follows: a Logical Algorithms transition $\sigma \overset{\text{LA}}{\rightarrowtail}_P \sigma'$ corresponds to an LTS transition $\sigma \xrightarrow{\alpha} \sigma'$ with $\alpha = \sigma' \backslash \sigma$, i.e., α represents the state change from σ to σ'. A CHR$^{\text{rp}}$ transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$ corresponds to an LTS transition $\mathsf{chr\_to\_la}(\sigma) \xrightarrow{\alpha} \mathsf{chr\_to\_la}(\sigma')$ with $\alpha = \mathsf{chr\_to\_la}(\sigma') \setminus \mathsf{chr\_to\_la}(\sigma)$ if this set is not empty and $\alpha = \tau$ otherwise.

Corollary 2 *The equality relation between the states of S_1 and S_2 is a weak bisimulation.*

5.4 Translating a subset of CHR$^{\text{rp}}$ into Logical Algorithms

In the previous section, we have shown that Logical Algorithms programs can be translated into equivalent CHR$^{\text{rp}}$ programs. In this section, we show how to do the opposite, i.e., how CHR$^{\text{rp}}$ programs can be translated into equivalent Logical Algorithms programs. This allows us to apply the meta-complexity theorem for Logical Algorithms to the translation of these CHR$^{\text{rp}}$ programs.

We need to impose some restrictions on the CHR$^{\text{rp}}$ programs that can be translated. These restrictions result from the fact that the Logical Algorithms language does not have the concept of an underlying constraint solver that offers both ask and tell built-in constraints. The following two properties are required:

1. In all *reachable* states $\sigma = \langle G, S, B, T\rangle_n$: $\mathsf{vars}(S) = \emptyset$. In words, all (stored) CHR constraints are ground.

2. All built-in constraints are comparisons; there are no built-in tell constraints.

The first property holds if the initial goal is ground and all rules are *variable restricted*, which means that all variables in the body of a rule, also appear in one of the rule heads. The second property implies that all reachable states are of the form $\langle G, S, \mathtt{true}, T\rangle_n$, i.e., the built-in constraint store is always equivalent to `true`.

To simplify the presentation, we also assume that the priority of dynamic priority rules is determined by the arguments of its left-most head. In general, we can use the transformation schema given in Section 5.2.1 to ensure that the resulting Logical Algorithms rules have the correct syntactical form.

5.4.1 Translation Schema

We now show how the rules of a CHR$^{\mathrm{rp}}$ program P are transformed into Logical Algorithms rules that form a program $T(P)$. To increase readability, we distinguish between simplification and simpagation rules on the one hand, and propagation rules on the other. A *simpagation* rule of the form

$$p :: r \ @ \ H_1, \ldots, H_{m-1} \backslash H_m, \ldots, H_n \iff g \mid B_1, \ldots, B_l$$

is transformed into

$$\begin{aligned} r' \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(H_m^{\mathsf{id}}), \ldots, \mathtt{del}(H_n^{\mathsf{id}}), \mathtt{del}(\mathtt{next\_id}(\mathit{Id}_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_l^{\mathsf{id}}, \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}} + l) \end{aligned}$$

where $H_i^{\mathsf{id}} = \mathtt{c}(\bar{X}, \mathit{Id}_i)$ if $H_i = \mathtt{c}(\bar{X})$, $B_i^{\mathsf{id}} = \mathtt{c}(\bar{X}, \mathit{Id}_{\mathsf{next}} + i - 1)$ if $B_i = \mathtt{c}(\bar{X})$ and $\mathit{Alldiff} = \{(\mathit{Id}_i \neq \mathit{Id}_j) \mid \mathcal{D} \models \bar{\exists}_{\emptyset} H_i = H_j \wedge g\}$. The disequalities in *Alldiff* are between the identifiers of those heads that are unifiable and for which the guard is still satisfiable after this unification. The case of a simplification rule is very similar. A propagation rule of the form

$$p :: r \ @ \ H_1, \ldots, H_n \implies g \mid B_1, \ldots, B_l$$

is transformed into the following two rules

$$\begin{aligned} r'_1 \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g \Rightarrow \mathtt{token}([\mathit{Id}_1, \ldots, \mathit{Id}_n], r) \\ r'_2 \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{token}([\mathit{Id}_1, \ldots, \mathit{Id}_n], r), \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(\mathtt{token}([\mathit{Id}_1, \ldots, \mathit{Id}_n], r)), \mathtt{del}(\mathtt{next\_id}(\mathit{Id}_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_l^{\mathsf{id}}, \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}} + l) \end{aligned}$$

where H_i^{id}, B_i^{id} and *Alldiff* are as before. The first of these rules generates a token. This token is removed by the second rule. The tokens are needed to prevent a given rule instance from firing more than once. Note that the transformation into two rules and the use of tokens does not increase the complexity compared to the original rule, as there is only one token for each combination of rule and constraint identifiers (as well as only one `next_id`/1 fact in any state).

The initial database consists of the goal (where each constraint is extended with a unique identifier) and a $\mathtt{next\_id}(\mathit{Id}_{\mathsf{next}})$ assertion (with $\mathit{Id}_{\mathsf{next}}$ the next free identifier).

Example 5.6 (Merge Sort) Listing 5.3 shows a CHR$^{\mathrm{rp}}$ implementation of the merge sort algorithm. Its input consists of a series of n (a power of 2) `number`/1 constraints. Its output is a sorted list of the numbers in the input, represented

```
1 :: ms1 @ arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).
2 :: ms2 @ merge(N,A),  merge(N,B) <=> A < B | merge(2*N+1,A), arrow(A,B).
3 :: ms3 @ number(X) <=> merge(0,X).
```

Listing 5.3: A CHR$^{\text{rp}}$ implementation of merge sort

```
ms1′ @ 1 : arrow(X,A,Id1), arrow(X,B,Id2), A < B, next_id(NId) =>
            del(arrow(X,B,Id2)), del(next_id(NId)),
            arrow(A,B,NId), next_id(NId+1).
ms2′ @ 2 : merge(N,A,Id1), merge(N,B,Id2), A < B, next_id(NId) =>
            del(merge(N,A,Id1)), del(merge(N,B,Id2)), del(next_id(NId)),
            merge(2*N+1,A,NId), arrow(A,B,NId+1), next_id(NId+2).
ms3′ @ 3 : number(X,Id), next_id(NId) => del(number(X,Id)),
            del(next_id(NId)), merge(0,X,NId), next_id(NId+1).
```

Listing 5.4: The Logical Algorithms translation of Listing 5.3

as `arrow`/2 constraints, where `arrow`(x, y) indicates that x is right before y. The Logical Algorithms translation is shown in Listing 5.4.

Note that in rules `ms1` and `ms2`, the guard prevents the constraints matching the heads from being equal, and so there are no disequality constraints between the CHR constraint identifiers in these rules. Using the Logical Algorithms meta-complexity result, we can derive that the total runtime of the translated merge sort algorithm is $\mathcal{O}(n \log n)$. A detailed analysis is given in Section 5.6.1 where we analyze the CHR$^{\text{rp}}$ implementation directly using a new meta-complexity theorem for CHR$^{\text{rp}}$. □

Example 5.7 (Less-or-Equal) We illustrate the translation of propagation rules by translating the `transitivity` rule from the `leq` program (Listing 3.1), shown again below for convenience. Its translation is shown in Listing 5.5.

```
2 :: transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

```
transitivity′1 @ 2 : leq(X,Y,Id1), leq(Y,Z,Id2), Id1 \= Id2 =>
                token([Id1,Id2],transitivity).
transitivity′2 @ 2 : leq(X,Y,Id1), leq(Y,Z,Id2), Id1 \= Id2,
        token([Id1,Id2],transitivity), next_id(NId) =>
                del(token([Id1,Id2],transitivity)), del(next_id(NId)),
                leq(X,Z,NId), next_id(NId+1).
```

Listing 5.5: LA translation of the `transitivity` rule from Listing 3.1

Note that since in the original rule, the two heads `leq`$(\mathtt{X}, \mathtt{Y})$ and `leq`$(\mathtt{Y}, \mathtt{Z})$ are unifiable (and there is furthermore no guard to prevent this from happening), we

have to add an explicit disequality between the constraint identifiers for these heads: $\mathtt{Id}_1 \neq \mathtt{Id}_2$. □

5.4.2 Correspondence between ω_p and LA Derivations

In this subsection, we prove that a CHR$^{\mathrm{rp}}$ program and its translation to Logical Algorithms are operationally equivalent. Again we introduce a mapping function:

$$\mathsf{la\_to\_chr}(\sigma) = \langle \emptyset, S, \mathtt{true}, T \rangle_n$$

where the CHR constraint store $S = \{c(\bar{X})\#Id \mid c(\bar{X}, Id) \in \sigma \wedge \mathtt{del}(c(\bar{X}, Id)) \notin \sigma\}$, the propagation history $T = \{\langle Ids, \epsilon, r\rangle \mid \mathtt{del}(\mathtt{token}(Ids, r)) \in \sigma\}$, and the next free identifier n is such that $\mathtt{next\_id}(n) \in \sigma$ and $\mathtt{del}(\mathtt{next\_id}(n)) \notin \sigma$. In the following, we consider a Logical Algorithms state σ reachable with respect to program $T(P)$ if it can be derived from an initial state consisting of CHR constraints extended with unique identifiers, and a single $\mathtt{next\_id}/1$ assertion with as argument the next free identifier. In this case, reachability amongst others implies that there can be only one (strictly) positive $\mathtt{next\_id}/1$ assertion in the database in any state, and no two CHR constraint representations share their identifier.

Theorem 9 *For every reachable LA state σ_i it holds that if $\sigma_i \overset{LA}{\rightarrowtail}_{T(P)} \sigma_j$, either $\mathsf{la\_to\_chr}(\sigma_i) = \mathsf{la\_to\_chr}(\sigma_j)$ or there exists a finite CHR derivation $\mathsf{la\_to\_chr}(\sigma_i) = \langle \emptyset, S, \mathtt{true}, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle C, S', \mathtt{true}, T'\rangle_n \overset{\omega_p}{\mapsto}^*_P \langle \emptyset, S'', \mathtt{true}, T'\rangle_{n'} = \mathsf{la\_to\_chr}(\sigma_j)$ consisting of an **Apply** transition, followed by zero or more **Introduce** transitions.*

Proof: Consider a transition $\sigma_i \overset{\mathrm{LA}}{\rightarrowtail}_{T(P)} \sigma_j$. The only type of transition in Logical Algorithms is the **Apply** transition which fires a rule. If $\mathsf{la\_to\_chr}(\sigma_i) = \mathsf{la\_to\_chr}(\sigma_j)$, then this rule must be of the form

$$r'_1 \ @\ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, Alldiff, g \Rightarrow \mathtt{token}([Id_1, \ldots, Id_m], r)$$

because all other types of rules either delete the representation of a CHR constraint which changes the CHR constraint store, or remove a token which results in an extended propagation history. We call the rule that fired a *token generation rule*.

Assume $\mathsf{la\_to\_chr}(\sigma_i) \neq \mathsf{la\_to\_chr}(\sigma_j)$ and the rule fired is of the form

$$\begin{aligned} r' \ @\ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, Alldiff, g, \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(H_l^{\mathsf{id}}), \ldots, \mathtt{del}(H_m^{\mathsf{id}}), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_o^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + o) \end{aligned}$$

which corresponds to a simplification ($l = 1$) or simpagation ($l > 1$) rule. We further assume the case of a simpagation rule; the case of a simplification rule is similar. If $r' \in T(P)$ (with $l > 1$), then P contains a rule

$$p :: r \ @\ H_1, \ldots, H_{l-1} \backslash H_l, \ldots, H_m \iff g \mid B_1, \ldots, B_o$$

Since the conditions for the Logical Algorithms **Apply** transition are satisfied, there exists a ground matching substitution θ such that for each antecedent $H_i^{\mathsf{id}} = c(\bar{X}, Id_i)$ $(1 \leq i \leq m)$ it holds that $\theta(H_i^{\mathsf{id}}) \in \sigma$ and $\mathtt{del}(\theta(H_i^{\mathsf{id}})) \notin \sigma$ and so by definition of the $\mathsf{la\_to\_chr}$ function, $\theta(H_i \# Id_i) \in S$ where $\mathsf{la\_to\_chr}(\sigma_i) = \sigma_i' = \langle \emptyset, S, \mathtt{true}, T \rangle_n$. For each comparison $g_i \in g$, it holds that $\mathcal{D} \models \theta(g_i)$ and so $\mathcal{D} \models \mathtt{true} \rightarrow \bar{\exists}_{\emptyset}\theta(g)$. Since r is a simpagation rule, the propagation history T does not contain any element of the form $\langle \_, \_, r \rangle$. Ignoring priorities for the moment, all conditions are satisfied such that the rule instance $\theta(r)$ can fire in state σ_i'. We return to the issue of priorities further on.

After firing $\theta(r)$ in state σ_i', the resulting state equals $\langle \theta(B_1 \wedge \ldots \wedge B_o), S', \mathtt{true}, T \rangle_n$ where $S' = S \backslash \{\theta(H_l \# Id_l), \ldots, \theta(H_m \# Id_m)\}$. In this state, the only applicable transition is the **Introduce** transition, which is applied o times before reaching a state with an empty goal. There are $o!$ possible orders in which the introductions can be applied, but the one we need is the order in which the B_i constraints appear in the rule body. Following this order, the state resulting from the introductions equals $\sigma_j' = \langle \emptyset, S'', \mathtt{true}, T \rangle_{(n+o)}$ where $S'' = S' \cup \{\theta(B_1)\#n, \ldots, \theta(B_o)\#(n+o-1)\}$. It is easy to see that this state σ_j' equals $\mathsf{la\_to\_chr}(\sigma_j)$, the state resulting from firing Logical Algorithms rule instance $\theta(r')$ in state σ_i.

If $\mathsf{la\_to\_chr}(\sigma_i) \neq \mathsf{la\_to\_chr}(\sigma_j)$ and the rule fired is not of the form shown above, then it must have the following form

$$\begin{aligned} r_2' \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{token}([Id_1, \ldots, Id_m], r), \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(\mathtt{token}([Id_1, \ldots, Id_m], r)), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_o^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + o) \end{aligned}$$

The corresponding CHR$^{\text{rp}}$ rule in P looks like

$$p :: r \ @ \ H_1, \ldots, H_m \implies g \mid B_1, \ldots, B_o$$

Again, since the conditions for the Logical Algorithms **Apply** transition are satisfied, there exists a ground matching substitution θ such that for each antecedent $H_i^{\mathsf{id}} = c(\bar{X}, Id_i)$ $(1 \leq i \leq m)$ in rule r_2' it holds that $\theta(H_i^{\mathsf{id}}) \in \sigma$ and $\mathtt{del}(\theta(H_i^{\mathsf{id}})) \notin \sigma$ and so by definition of the $\mathsf{la\_to\_chr}$ function, $\theta(H_i \# Id_i) \in S$ where $\mathsf{la\_to\_chr}(\sigma_i) = \sigma_i' = \langle \emptyset, S, \mathtt{true}, T \rangle_n$. For each comparison $g_i \in g$, it holds that $\mathcal{D} \models \theta(g_i)$ and so $\mathcal{D} \models \mathtt{true} \rightarrow \bar{\exists}_{\emptyset}\theta(g)$. The propagation history T cannot contain $\langle \theta([Id_1, \ldots, Id_m]), \epsilon, r \rangle$ because by definition of the $\mathsf{la\_to\_chr}$ function this would imply that the atom $\mathtt{token}(\theta([Id_1, \ldots, Id_m), r)$ was deleted in some earlier state, which contradicts with the applicability of rule instance $\theta(r_2')$. If we again ignore the issue of priorities, all conditions are satisfied such that $\theta(r)$ can fire in state σ_i'.

After firing $\theta(r)$ in state σ_i', the resulting state equals $\langle \theta(B_1 \wedge \ldots \wedge B_o), S, \mathtt{true}, T' \rangle_n$ where $T' = T \cup \{\langle [Id_1, \ldots, Id_m], \epsilon, r \rangle\}$. In this state, the only applicable transition is the **Introduce** transition, which is applied o times before reaching a state with an empty goal. Given again that these introductions are applied in the

order in which the B_i constraints appear in the rule body, then the resulting state equals $\sigma'_j = \langle \emptyset, S', \mathtt{true}, T' \rangle_{(n+o)}$ where $S' = S \cup \{\theta(B_1)\#n, \ldots, \theta(B_o)\#(n+o-1)\}$. It is again easy to see that this state σ'_j equals $\mathsf{la\_to\_chr}(\sigma_j)$, the state resulting from firing Logical Algorithms rule instance $\theta(r'_2)$ in state σ_i.

This proves the theorem if we ignore priorities. Theorem 10 (see next) shows that each CHR$^{\mathrm{rp}}$ rule firing has a corresponding Logical Algorithms rule firing. Under the assumption that this theorem also holds ignoring rule priorities, we have that the highest priority rule instances are the same in both programs given corresponding states and ignoring token generation rules. □

Theorem 10 *For every reachable CHR^{rp} state σ_i and reachable Logical Algorithms state σ'_i with $\mathsf{la\_to\_chr}(\sigma'_i) = \sigma_i$, there exists a finite Logical Algorithms derivation $\sigma'_i \overset{LA}{\longmapsto}{}^*_{T(P)} \sigma'_{i^*}$ with $\mathsf{la\_to\_chr}(\sigma'_{i^*}) = \sigma_i$ such that if $\sigma_i = \langle \emptyset, S, \mathtt{true}, T \rangle_n \overset{\omega_p}{\longmapsto}_P \langle C, S', \mathtt{true}, T' \rangle_n \overset{\omega_p}{\longmapsto}{}^*_P \langle \emptyset, S'', \mathtt{true}, T' \rangle_{n'} = \sigma_j$ where the derivation consists of a single **Apply** transition, followed by zero or more **Introduce** transitions, then $\sigma'_{i^*} \overset{LA}{\longmapsto}_{T(P)} \sigma'_j$ with $\mathsf{la\_to\_chr}(\sigma'_j) = \sigma_j$ and if σ_i is a final state then σ'_{i^*} is also a final state.*

Proof: Let there be given a reachable LA state σ'_i with $\mathsf{la\_to\_chr}(\sigma'_i) = \sigma_i$. Because of Theorem 9, state σ_i is also reachable in CHR$^{\mathrm{rp}}$ with respect to program P. Assume $\sigma_i = \langle \emptyset, S, \mathtt{true}, T \rangle_n \overset{\omega_p}{\longmapsto}_P \langle C, S', \mathtt{true}, T' \rangle_n \overset{\omega_p}{\longmapsto}{}^*_P \langle \emptyset, S'', \mathtt{true}, T' \rangle_{n'} = \sigma_j$ where the derivation consists of a single **Apply** transition, followed by zero or more **Introduce** transitions, and let $\theta(r)$ be the CHR$^{\mathrm{rp}}$ rule instance that fired in state σ_i. If r is simplification ($l = 1$) or simpagation ($l > 1$) rule

$$p :: r \ @ \ H_1, \ldots, H_{l-1} \backslash H_l, \ldots, H_m \iff g \mid B_1, \ldots, B_o$$

then $\theta(H_i)\#id_i \in S$ for $1 \leq i \leq m$ with $id_i \neq id_j$ if $i \neq j$, and $\mathcal{D} \models \bar{\exists}_\emptyset \theta(g)$. Furthermore, $T(P)$ contains a rule

$$\begin{aligned} r' \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(H_l^{\mathsf{id}}), \ldots, \mathtt{del}(H_m^{\mathsf{id}}), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_o^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + o) \end{aligned}$$

Now let θ' be a ground matching substitution such that $\theta'|_{\mathsf{vars}(\theta)} = \theta$ where $\theta'|_{\mathsf{vars}(\theta)}$ is the projection of θ' on the variables in θ, and such that both $\theta'(Id_i) = id_i$ for $1 \leq i \leq m$ and $\theta'(Id_{\mathsf{next}}) = n$. Since for $1 \leq i \leq m$, $H_i^{\mathsf{id}} = c(\bar{X}, Id_i)$ if $H_i = c(\bar{X})$, it holds that $\theta'(H_i^{\mathsf{id}}) \in \sigma'_i$ and $\mathtt{del}(\theta'(H_i^{\mathsf{id}})) \notin \sigma'_i$. Also, $\mathcal{D} \models \bar{\exists}_\emptyset \theta(g)$ implies $\mathcal{D} \models \theta(g_i)$ for each comparison $g_i \in g$.[7] The *Alldiff* conditions hold because $\theta'(Id_i) = \theta'(Id_j)$ implies that $i = j$. Because of the reachability of state σ'_i,

[7] Because $\theta(g)$ is ground, there is no existential quantification.

there is exactly one strictly positive `next_id`/1 assertion in σ'_i whose argument equals n. Finally, the rule conclusion cannot be already included in the state σ'_i because it includes amongst others the deletion of at least one of the antecedents. Ignoring priorities, all conditions are satisfied such that rule instance $\theta'(r')$ can fire in state σ'_i, resulting in a state $\sigma'_j = \mathsf{la\_to\_chr}(\sigma_j)$. As stated earlier in the proof of Theorem 9, the combination of Theorems 9 and 10 without taking into account the priorities, implies that the highest priority applicable rule instances are the same in corresponding states, ignoring token generation rules.

Now assume that in the CHR$^{\text{rp}}$ state σ_i, a rule instance $\theta(r)$ fires where r is a propagation rule:

$$p :: r \ @ \ H_1, \ldots, H_m \implies g \mid B_1, \ldots, B_o$$

In this case the Logical Algorithms translation $T(P)$ contains the following rules:

$$\begin{aligned} r'_1 \ @ \ p : {} & H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{All diff}, g \Rightarrow \mathtt{token}([Id_1, \ldots, Id_n], r) \\ r'_2 \ @ \ p : {} & H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{All diff}, g, \mathtt{token}([Id_1, \ldots, Id_n], r), \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \\ & \mathtt{del}(\mathtt{token}([Id_1, \ldots, Id_n], r)), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ & B_1^{\mathsf{id}}, \ldots, B_l^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + l) \end{aligned}$$

A similar analysis as above shows that there exists a matching substitution θ' with $\theta'|_{\mathsf{vars}(\theta)} = \theta$ and both $\theta'(Id_i) = id_i$ for $1 \leq i \leq m$ and $\theta'(Id_{\mathsf{next}}) = n$, such that rule instance $\theta'(r'_1)$ can fire (ignoring priorities) if $\mathtt{token}([id_1, \ldots, id_n], r) \notin \sigma'_i$ and $\theta'(r'_2)$ otherwise. If $\theta'(r'_1)$ fires then the resulting state σ'_{i^*} equals $\sigma'_i \cup \{\mathtt{token}([id_1, \ldots, id_n], r)\}$ and clearly $\mathsf{la\_to\_chr}(\sigma'_{i^*}) = \mathsf{la\_to\_chr}(\sigma'_i)$. Moreover, in state σ'_{i^*}, rule instance $\theta'(r'_2)$ can fire and for the resulting state σ'_j it holds that $\mathsf{la\_to\_chr}(\sigma'_j) = \sigma_j$. If already $\mathtt{token}([id_1, \ldots, id_n], r) \in \sigma'_i$ then the same reasoning holds with $\sigma'_i = \sigma'_{i^*}$.

Finally, assume that CHR$^{\text{rp}}$ state σ_i is a final state. If σ'_i is not a final Logical Algorithms state, then because of Theorem 9, the only applicable rules are those that do not change the result of the $\mathsf{la\_to\_chr}$ function. Only the token generation rules satisfy this property. Since they only generate tokens and these tokens do not appear in their antecedents, these rules can fire only finitely many times before a final Logical Algorithms state σ_{i^*} is reached. □

5.5 Implementing CHR$^{\text{rp}}$, the Logical Algorithms Way

This section presents a new implementation for CHR$^{\text{rp}}$, based on the implementation proposal for Logical Algorithms by Ganzinger and McAllester (2002), as well as on the scheduling algorithm presented in (De Koninck 2007). The purpose of

this implementation is not to replace our existing CHR^{rp} implementation as presented in Chapter 3, but to support a new meta-complexity theorem for CHR^{rp}, based on the result for Logical Algorithms, and extended towards the full CHR^{rp} language. This includes in particular support for non-ground constraints and a built-in constraint theory. We note that a better worst case complexity for certain operations is not always worthwhile in practice due to larger constant factors in the average case. Also, the proposed implementation may not always achieve a better complexity than the existing implementation. The main purpose remains to have a relatively straightforward way to derive for a given CHR^{rp} program, a bound that is guaranteed to be an upper-bound for at least the implementation proposed. Since the meta-complexity result is insensitive to constant factors, we can present the new implementation as a source-to-source transformation to regular CHR.

The proposed implementation consists of the compilation of the CHR^{rp} rules of the input program into regular CHR rules in which matching is made explicit, combined with a scheduler module that is responsible for the execution control. The implementation is correct if it is executed according to the refined operational semantics of CHR (see Section 2.2.2). We have based our implementation on the high-level implementation proposal for Logical Algorithms by Ganzinger and McAllester (2002), extended where necessary to support general built-in constraints. By using a CHR implementation with advanced indexing support, like for example the K.U.Leuven CHR system (Schrijvers and Demoen 2004), our implementation also offers strong complexity guarantees that facilitate a new meta-complexity theorem for CHR^{rp}, similar to the one for Logical Algorithms (see Section 5.6). In the following, we make use of Prolog as CHR's host language, but the implementation can easily be adapted to work with a different host language.

5.5.1 Overview

The implementation is based on a form of lazy (on-demand) matching with retainment of previously computed partial matches. It combines the concept of alpha and beta memories from the RETE algorithm (Forgy 1982), with lazy matching as for example implemented by the LEAPS algorithm (Miranker et al. 1990).[8] The basic idea is as follows. A new constraint can function both as a single headed partial or full match, and as an extension of an existing partial match into either a new (larger) partial match or a full match. In order to extend partial matches, all previously computed matches are stored. A scheduler decides which partial match is extended with which constraint, or which full match has its corresponding rule instance fired. More details on the scheduler are given in Section 5.5.3.

First, to simplify the presentation, we propose an alternative syntax for CHR^{rp}

[8] Most current CHR systems, including the K.U.Leuven CHR system and the CHR^{rp} system of Chapter 3, use a variant of the LEAPS algorithm for rule matching.

rules. An *intermediate form* CHR$^{\mathrm{rp}}$ rule looks as follows:

$$p :: r \ @ \ s_1 A_1, \ldots, s_n A_n \iff B$$

where $s_i \in \{+, -, ?\}$ and A_i is an atom for $1 \leq i \leq n$. If $s_i = +$ or $s_i = -$ then A_i must be a CHR constraint and if $s_i =?$ then A_i must be a built-in constraint. An intermediate form CHR$^{\mathrm{rp}}$ rule corresponds to a regular CHR$^{\mathrm{rp}}$ rule as follows: a term $+A$ corresponds to a kept head A, a term $-A$ corresponds to a removed head A, and a term $?A$ corresponds to a conjunct of the rule guard. The main advantage of the intermediate form is that it supports specifying a join order for the heads, as well as an evaluation order for the guards. In particular it supports specifying the evaluation of part of the guard after having computed only a partial rule match. The intermediate form gives us the same syntactical flexibility as exists in the Logical Algorithms language where comparisons are interleaved with the (kept and removed) user-defined antecedents.

Consider, in general, a simpagation rule of the form

$$p :: r \ @ \ H_1, \ldots, H_i \backslash H_{i+1}, \ldots, H_n \iff g \mid B$$

where the guard g is a conjunction of atomic guards $g_1, \ldots, g_m$. We can rewrite this rule in intermediate form syntax (amongst others) as follows:

$$p :: r \ @ \ +H_1, \ldots, +H_i, -H_{i+1}, \ldots, -H_n, ?g_1, \ldots, ?g_m \iff B$$

In the following, we assume that all rules have the following form

$$p :: r \ @ \ \pm H_1, ?g_1, \pm H_2, ?g_2, \ldots, \pm H_n, ?g_n \iff B$$

where $\pm$ means $+$ or $-$. Each g_i $(1 \leq i \leq n)$ can be a conjunction of primitive built-in constraints, and can in particular also be equal to `true`. The transformation from regular CHR$^{\mathrm{rp}}$ syntax to intermediate form syntax can be done automatically using the above transformation schema, or by hand.

Using terminology similar to that of Ganzinger and McAllester (2002), we refer to a partial match, matching the heads $H_1, \ldots, H_i$ and satisfying the partial guard $g_1 \wedge \ldots \wedge g_{i-1}$, as a *suspended strong prefix firing*. If also the partial guard g_i is satisfied, we speak of a *regular* (or *non-suspended*) strong prefix firing. A constraint matching the next head H_{i+1} is called a *prefix extension* of such a (regular) strong prefix firing. A prefix firing that consists of all heads is (also) called a (suspended or regular) *rule firing*.[9] Every prefix firing contains the left-most head and hence determines the rule priority. In our implementation, we assume that all guards are monotone, i.e., once they are entailed by the built-in constraint store, they remain entailed in any later state. This is in fact required by the CHR operational (and declarative) semantics, although most current CHR systems also support non-monotone (impure) guards like for example `var`/1 in CHR on top of Prolog.

[9] A rule firing actually means a rule instance that is applicable. To avoid confusion, we refer to the actual firing of such a rule firing as *firing a rule instance*.

5.5.2 Program-Dependent Part

The program-dependent part of our implementation (i.e., the part that depends on the actual program to be implemented) consists of rules for

- generating a representation for CHR constraint occurrences and deleting them when the represented constraint is removed;
- generating and scheduling constraints representing prefix firings, prefix extensions and rule firings and deleting them when a constituent constraint is removed;
- matching prefix firings with prefix extensions, firing rule instances, and managing suspended prefix and rule firings.

The different types of rules of the program-dependent part are illustrated by using a running example program, namely Dijkstra's shortest path algorithm, already given in the Logical Algorithms language in Example 5.2 and given here in CHR^{rp} intermediate form syntax. To illustrate non-trivial head matching, we have added a rule `d1` that removes simple loops from the input graph.

```
    1 :: d1 @ -e(V,_,V),   ?true                          <=> true.
    1 :: d2 @ +source(V),  ?true                          <=> dist(V,0).
    1 :: d3 @ -dist(V,D1), ?true, +dist(V,D2), ?(D2 < D1) <=> true.
D + 2 :: d4 @ +dist(V,D),  ?true, +e(V,C,U),   ?true      <=> dist(U,D+C).
```

Constraint Occurrence Representation

Although CHR^{rp} constraints and CHR constraints obviously have the same syntax and semantics (i.e., multi-set semantics with non-monotone deletion), we introduce a new representation for them to allow unambiguous reference, reduce work in case of constraint reactivation, and support the efficient deletion of those prefix firings, prefix extensions, and rule firings in which they participate (see further). For each CHR^{rp} constraint of predicate c/n, we create a set of unique *occurrence representations* $c\_occ\_i/(n+1)$, one for each occurrence of the predicate in a rule head. The arguments of a $c\_occ\_i/(n+1)$ constraint consist of the arguments of the original c/n constraint, together with a unique constraint identifier that is shared by all occurrence representations. This identifier is an uninstantiated variable as long as the constraint is in the store and is instantiated the moment that the constraint is to be deleted. For each user-defined constraint predicate c/n with m occurrences, the occurrence representations are generated using rules of the following form.

c`(X1,...,Xn) <=>` c`_occ_1(X1,...,Xn,Id), ...,` c`_occ_`m`(X1,...,Xn,Id).`

For the example program, these rules look as follows.

```
source(V) <=> source_occ_1(V,Id).
dist(V,D) <=> dist_occ_1(V,D,Id), dist_occ_2(V,D,Id), dist_occ_3(V,D,Id).
e(V,C,U)  <=> e_occ_1(V,C,U,Id),  e_occ_2(V,C,U,Id).
```

RETE Memory Constraints

Regular and suspended prefix firings as well as prefix extensions are represented as CHR constraints. We call them RETE memory constraints because they coincide with the alpha and beta memories of the RETE algorithm. The RETE memory constraints contain all arguments of their constituent CHR constraints, as well as their identifiers. Each RETE memory constraint moreover has its own unique identifier. We use the following functors for RETE memory constraints:

- r`_pf_`i for a regular (non-suspended) prefix firing of rule r, consisting of i heads, and r`_pf_`i`_suspended` for its suspended version
- r`_pe_`i for a prefix extension, consisting of the $(i+1)^{\text{th}}$ head of rule r
- r`_rf` for a (regular) rule firing of rule r and r`_rf_suspended` for its suspended version.

If in a rule r, the partial guard after the i^{th} head equals `true`, then there is no suspended version of the i-headed prefix firings of r, or of its rule firings if r is an i-headed rule. In the example program, the following prefix firings, prefix extensions and rule firings are defined:

- `d1_rf`/4
- `d2_rf`/3
- `d3_pf_1`/4, `d3_pe_1`/3, `d3_rf`/6 and `d3_rf_suspended`/6
- `d4_pf_1`/4, `d4_pe_1`/4 and `d4_rf`/7

Suspended Prefix and Rule Firings

Suspended prefix and rule firings are converted into regular prefix and rule firings as soon as the relevant part of the guard is entailed. If on the other hand this partial guard is disentailed, the suspended prefix or rule firing is removed. Given a rule in intermediate form syntax

$$p :: r \ @ \ \pm H_1, ?g_1, \pm H_2, ?g_2, \ldots, \pm H_n, ?g_n \iff B$$

we generate the following rules:

- For each i-headed suspended prefix firing:

```
r_pf_i_suspended(X1,...,Xm,Id1,...,Idi,SId) <=>
            gi | r_pf_i(X1,...,Xm,Id1,...,Idi,SId),
            schedule_pf(r_i(Y1,...,Yl),p,SId).
r_pf_i_suspended(X1,...,Xm,Id1,...,Idi,SId) <=> \+ gi | true.
```

where $\mathtt{Y}_1,\ldots,\mathtt{Y}_l$ are those variables in $\mathtt{X}_1,\ldots,\mathtt{X}_m$ that also appear in H_{i+1}

- For each rule firing:

```
r_rf_suspended(X1,...,Xm,Id1,...,Idn,SId) <=> gn |
            r_rf(X1,...,Xm,Id1,...,Idn,SId), schedule_rf(p,SId).
r_rf_suspended(X1,...,Xm,Id1,...,Idn,SId) <=> \+ gn | true.
```

Note that if g_i or g_n equals `true`, then we can apply unfolding to replace occurrences of respectively r`_pf_`i`_suspended`$/(m+i+1)$ and r`_rf_suspended`$/(m+n+1)$ by the bodies of the corresponding rules above (see (Tacchella et al. 2007)). After this unfolding step, some of the above rules may be removed. In the example program, only a rule firing of rule `d3` can be suspended.

```
d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=>
        D2 < D1 | d3_rf(V,D1,D2,Id1,Id2,SId), schedule_rf(1,SId).
d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=> \+ (D2 < D1) | true.
```

In each second rule, `\+` C is a safe approximation of the negation of constraint C, i.e., it is only entailed if constraint C cannot possibly hold. In the Prolog context, the built-in negation as failure can be used.

Suspended constraints are attached to all guarded variables so that they are reactivated whenever one of these variables is affected by a built-in constraint. We assume that both attaching and detaching can be done in constant time, although certain current CHR implementations like the K.U.Leuven CHR system do not support detaching in constant time.

Scheduling

Each constraint occurrence corresponds to a (potentially suspended) rule firing if it is the only head of a single-headed rule, a (potentially suspended) prefix firing if it is the first head of a multi-headed rule, and a prefix extension in all other cases. A conversion between constraint occurrence and rule firing, prefix firing or prefix extension is made as soon as the constraint in question matches with the head. If such a match is shown to be impossible, the constraint occurrence is discarded. Let there be given a head constraint $c(X_1,\ldots,X_n)$. The following function is used

to construct a head match.

$$\mathsf{head\_match}([X|\bar{X}]) = \begin{cases} \langle [X|\bar{Y}], g\rangle \text{ if } X \text{ is a variable and } X \notin \mathsf{vars}(\bar{X}) \\ \langle [Y|\bar{Y}], (Y = X) \wedge g\rangle \text{ otherwise} \end{cases}$$
$$\text{where } \langle \bar{Y}, g\rangle = \mathsf{head\_match}(\bar{X})$$
$$\mathsf{head\_match}(\epsilon) = \langle \epsilon, \mathtt{true}\rangle$$

Now, for each rule in intermediate form syntax

$$p :: r \ @ \ \pm H_1, ?g_1, \pm H_2, ?g_2, \ldots, \pm H_n, ?g_n \iff B$$

and for $1 \leq i \leq n$ we generate the rules below where $H_i = c(X'_1, \ldots, X'_n)$ is the j^{th} occurrence of the user-defined constraint predicate c/n, $\langle [X_1, \ldots, X_n], g\rangle = \mathsf{head\_match}([X'_1, \ldots, X'_n])$, and $\{Y_1, \ldots, Y_m\} = \mathsf{vars}(H_i) \setminus \mathsf{vars}(\{H_1, \ldots, H_{i-1}\})$.

- If $i = n = 1$:

  $c\_occ\_j(X_1,\ldots,X_n,\mathtt{Id})$ `<=>` g `|` $r\_rf\_suspended(Y_1,\ldots,Y_m,\mathtt{Id},\mathtt{SId})$.
  $c\_occ\_j(X_1,\ldots,X_n,\mathtt{Id})$ `<=> \+` g `|` `true`.

- If $i = 1$ and $n > 1$:

  $c\_occ\_j(X_1,\ldots,X_n,\mathtt{Id})$ `<=>` g `|` $r\_pf\_1\_suspended(Y_1,\ldots,Y_m,\mathtt{Id},\mathtt{SId})$.
  $c\_occ\_j(X_1,\ldots,X_n,\mathtt{Id})$ `<=> \+` g `|` `true`.

- Otherwise, if $i > 1$:

  $c\_occ\_j(X_1,\ldots,X_n,\mathtt{Id})$ `<=>` g `|` $r\_pe\_i-1(Y_1,\ldots,Y_m,\mathtt{Id},\mathtt{SId})$,
  `schedule_pe`$(r\_i-1(Z_1,\ldots,Z_l),\mathtt{SId})$.
  $c\_occ\_j(X_1,\ldots,X_n,\mathtt{Id})$ `<=> \+` g `|` `true`.

  where $\{Z_1, \ldots, Z_l\} = \mathsf{vars}(H_i) \cap \mathsf{vars}(\{H_1, \ldots, H_{i-1}\})$.

In the above, if $g =$ `true` then the second rule of each pair of rules can be discarded. The suspended prefix and rule firings can sometimes be replaced by regular prefix and rule firings by unfolding.

In the example program, only the first occurrence of the `e`/3 constraint has a non-trivial head match (the first and last argument must be the same). All single-headed prefix and rule firings are followed by the trivial guard `true` and so we only generate regular prefix and rule firings. They are scheduled using the `schedule_pf`/3 and `schedule_rf`/2 predicates.

```
source_occ_1(V,Id) <=> d2_rf(V,Id,SId), schedule_rf(1,SId).

dist_occ_1(V,D,Id) <=> d3_pf_1(V,D,Id,SId), schedule_pf(d3_1(V),1,SId).
dist_occ_2(V,D,Id) <=> d3_pe_1(D,Id,SId), schedule_pe(d3_1(V),SId).
dist_occ_3(V,D,Id) <=> d4_pf_1(V,D,Id,SId), schedule_pf(d4_1(V),D+2,SId).
```

```
e_occ_1(V,C,U,Id) <=> V = U | d1_rf(V,C,Id,SId), schedule_rf(1,SId).
e_occ_1(V,C,U,Id) <=> \+ (V = U) | true.
e_occ_2(V,C,U,Id) <=> d4_pe_1(C,U,Id,SId), schedule_pe(d3_1(V),SId).
```

Prefix firings and extensions are scheduled using a key containing their shared variables. For example for the prefix firings consisting of the first head of rule `d3` and the corresponding prefix extensions consisting of the second head of the same rule, the key equals `d3_1(V)`.

Similar to the suspended prefix and rule firings, the constraint occurrences are attached to all guarded variables. We again assume that both attaching and detaching can be done in constant time.

Matching and Firing

The scheduler initiates the firing of a rule instance by asserting a `fire`/1 constraint, and the matching of a prefix firing with a prefix extension by asserting a `match`/2 constraint. These constraints have as arguments the identifiers of the corresponding RETE memory constraints. After matching a prefix firing with a prefix extension, a new suspended prefix or rule firing is generated. For a given n-headed rule r with $n > 1$ and for $1 \leq i \leq n-2$, we generate the following rule

```
r_pf_i(X1,...,Xm,Id1,...,Idi,SId1), r_pe_i(Xm+1,...,Xl,Idi+1,SId2) \
        match(SId1,SId2) <=> Idi+1 \== Id1, ..., Idi+1 \== Idi |
                r_pf_i+1_suspended(X1,...,Xl,Id1,...,Idi+1).
```

and similarly for $i = n-1$:

```
r_pf_n-1(X1,...,Xm,Id1,...,Idn-1,SId1), r_pe_n-1(Xm+1,...,Xl,Idn,SId2) \
        match(SId1,SId2) <=> Idn \== Id1, ..., Idn \== Idn-1 |
                r_rf_suspended(X1,...,Xl,Id1,...,Idn).
```

A rule firing of an n-headed rule r with body B is fired as follows:

```
r_rf_i(X1,...,Xm,Id1,...,Idn,SId), fire(SId) <=>
                Idr(1) = dead, ..., Idr(l) = dead, B.
```

where $r(1), \ldots, r(l)$ are the indices of the removed heads of the rule (if any). We furthermore add the following rules at the end of the code, to make sure the CHR compiler detects that the `match`/2 and `fire`/1 constraints are never to be stored.

```
match(_,_) <=> true.
fire(_) <=> true.
```

For the example program, the generated code is as follows:

```
d1_rf(V,C,Id,SId), fire(SId) <=> Id = dead.
d2_rf(V,Id,SId), fire(SId) <=> dist(V,0).
d3_pf_1(V,D1,Id1,SId1), d3_pe_1(D2,Id2,SId2) \ match(SId1,SId2) <=>
        Id2 \== Id1 | d3_rf_suspended(V,D1,D2,Id1,Id2,SId).
d3_rf(V,D1,D2,Id1,Id2,SId), fire(SId) <=> Id1 = dead.
d4_pf_1(V,D,Id1,SId1), d4_pe_1(C,U,Id2,SId2) \ match(SId1,SId2) <=>
        Id2 \== Id1 | d4_pf(V,D,C,U,Id1,Id2,SId), schedule_rf(D+2,SId).
d4_rf(V,D,C,U,Id1,Id2,SId), fire(SId) <=> dist(U,D+C).

match(_,_) <=> true.
fire(_) <=> true.
```

Clean-up

Whenever a constraint's identifier variable is instantiated, its occurrence representations, as well as those RETE memory constraints in which it participates, are removed. The rules look as follows.

- For the i^{th} occurrence representation for constraint predicate c/n:

  c_occ_i(X1,...,Xn,Id) <=> nonvar(Id) | true.

- For an i-headed suspended prefix firing of rule r:

  r_pf_i_suspended(X1,...,Xm,Id1,...,Idi,SId) <=> nonvar(Id1) | true.
  ...
  r_pf_i_suspended(X1,...,Xm,Id1,...,Idi,SId) <=> nonvar(Idi) | true.

- For an i-headed regular prefix firing of rule r:

  r_pf_i(X1,...,Xm,Id1,...,Idi,SId) <=> nonvar(Id1) | remove_pf(SId).
  ...
  r_pf_i(X1,...,Xm,Id1,...,Idi,SId) <=> nonvar(Idi) | remove_pf(SId).

- For a prefix extension of an i-headed prefix firing of rule r:

  r_pe_i(X1,...,Xm,Id,SId) <=> nonvar(Id) | remove_pe(SId).

- For a suspended rule firing of an n-headed rule r:

  r_rf_suspended(X1,...,Xm,Id1,...,Idn,SId) <=> nonvar(Id1) | true.
  ...
  r_rf_suspended(X1,...,Xm,Id1,...,Idn,SId) <=> nonvar(Idn) | true.

- For a regular rule firing of an n-headed rule r:

```
r_rf(X1,...,Xm,Id1,...,Idn,SId) <=> nonvar(Id1) | remove_rf(SId).
...
r_rf(X1,...,Xm,Id1,...,Idn,SId) <=> nonvar(Idn) | remove_rf(SId).
```

The predicates `remove_pf`/1, `remove_pe`/1 and `remove_rf`/1 remove respectively a prefix firing, prefix extension and rule firing from the schedule. The following clean-up rules are generated for the example program.

```
source_occ_1(V,Id) <=> nonvar(Id) | true.
dist_occ_1(V,D,Id) <=> nonvar(Id) | true.
dist_occ_2(V,D,Id) <=> nonvar(Id) | true.
dist_occ_3(V,D,Id) <=> nonvar(Id) | true.
e_occ_1(V,C,U,Id)  <=> nonvar(Id) | true.
e_occ_2(V,C,U,Id)  <=> nonvar(Id) | true.

d1_rf(V,C,Id,SId)          <=> nonvar(Id)  | remove_rf(SId).
d2_rf(V,Id,SId)            <=> nonvar(Id)  | remove_rf(SId).
d3_pf_1(V,D1,Id1,SId)      <=> nonvar(Id1) | remove_pf(SId).
d3_pe_1(D2,Id2,SId)        <=> nonvar(Id2) | remove_pe(SId).
d3_rf(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id1) | remove_rf(SId).
d3_rf(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id2) | remove_rf(SId).
d4_pf_1(V,D,Id1,SId)       <=> nonvar(Id1) | remove_pf(SId).
d4_pe_1(C,U,Id2,SId)       <=> nonvar(Id2) | remove_pe(SId).
d4_rf(V,D,C,U,Id1,Id2,SId) <=> nonvar(Id1) | remove_rf(SId).
d4_rf(V,D,C,U,Id1,Id2,SId) <=> nonvar(Id2) | remove_rf(SId).

d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id1) | true.
d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id2) | true.
```

5.5.3 Program-Independent Part: The Scheduler

The scheduler implements the `schedule_rf`/2, `schedule_pf`/3 and `schedule_pe`/2 predicates. It furthermore implements the `execute`/0 predicate which retrieves and executes the highest priority scheduled task. This task either is the firing of a rule instance by asserting a `fire`/1 constraint, or the matching of a prefix firing with a prefix extension by asserting a `match`/2 constraint. The `execute`/0 predicate recursively calls itself until no more tasks are scheduled. It is first called after processing the initial goal.

For the implementation of the scheduler, we use a variant of the scheduling algorithm presented in (De Koninck 2007). In this work, we present a data structure representing *schedules* which are sets of *elements* and *nodes*. It supports the following operations: creating a new schedule, adding a new element or a new node to a given schedule, deleting an element or a node from its schedule, merging two schedules (after which the resulting schedule contains the elements and nodes of both original schedules), and finally, generating a new *match*. A match

is the combination of an element and a node, both of which belong to the same schedule. The algorithm ensures that no match is generated twice. Furthermore, it only fails to generate a new match if all elements and nodes belonging to any single schedule have already been matched. The algorithm ensures that all defined operations take quasi-constant time.

We can use our algorithm to maintain which prefix firings are still to match with which prefix extensions. Here, a prefix firing is mapped to a schedule's element, and a prefix extension is mapped to a schedule's node. A schedule of (De Koninck 2007) roughly corresponds to an $\mathcal{W}(r,t)$ data structure of (Ganzinger and McAllester 2002). These $\mathcal{W}(r,t)$ data structures consist of a series (implemented as a linear linked list) of *prefix blocks*, which are sets of prefix firings and (apart from the last one) are associated with a prefix extension.

The semantics of the $\mathcal{W}(r,t)$ data structure is that the prefix firings of a given prefix block are still to match with the prefix extension associated to it, as well as with all prefix extensions associated to subsequent prefix blocks. The last prefix block has no associated prefix extension, and represents those prefix firings that have been matched with all prefix extensions and hence are passive (or *completed* using the terminology of Ganzinger and McAllester (2002)). Whenever a prefix extension is deleted, its prefix block is merged with the next prefix block.

There is one $\mathcal{W}(r,t)$ data structure for each prefix length of each rule and for each combination of arguments shared between a prefix firing and prefix extension. Each prefix block is represented as a (local) priority queue whose items are the block's prefix firing. The highest priority item of each prefix block, together with its associated prefix extension, is also represented in a global priority queue. This prefix block representative is updated whenever the highest priority prefix firing of the prefix block is removed, a new prefix firing has the highest priority, or the associated prefix extension is removed. The global priority queue furthermore contains a representative for each rule firing. The reason for using two layers of priority queues is to reduce the amount of work needed when the prefix firings of a prefix block all become passive due to a prefix extension removal. It is the global priority queue that determines the next task to perform, i.e., matching a prefix firing with a prefix extension, or firing a rule instance.

In the context of CHR$^{\mathrm{rp}}$, built-in constraint (in particular equality constraints) on the arguments shared between a prefix firing and extension, may require merging of $\mathcal{W}(r,t)$ data structures. The data structure of De Koninck (2007) supports schedule merges in quasi constant time. The most notable difference with the $\mathcal{W}(r,t)$ data structure of Ganzinger and McAllester (2002) is that the prefix blocks form a circular linked list. Using this representation, merging schedules consists of cross-linking the circular lists and reactivating the prefix firings that were passive before the merge. Special care is taken to prevent both that a prefix firing is being matched with the same prefix extension more than once, and that a prefix firing 'misses' a prefix extension.

One consequence of using a circular linked list instead of a linear one to represent the prefix blocks, is that it is unclear (or more precisely, too expensive to decide) which prefix firings become passive whenever a prefix extension is deleted. Therefore, this decision is postponed until the scheduler tries to match the prefix firing with the next prefix extension in line. For complexity reasons, it is important that all prefix firings that have simultaneously been reactivated, and have not been matched with a prefix extension since this reactivation, are simultaneously made passive in time independent of the number of prefix firings affected. In (De Koninck 2007), a so-called *element schedule* based on a stack is proposed to supports this. In our context, we need an element schedule that is based on priority queues. It works as follows.

We use three types of priority queues. The first one is a single *global* priority queue which contains an item for each rule firing, for each *active* prefix firing that either has not been passive before or has been matched with at least one prefix extension since its last activation, and finally, for each set of prefix firings that have been simultaneously activated and have not been matched with a prefix extension since. A second type of priority queues is called a *local* queue and represents the above mentioned sets of prefix firings. Finally, the third type of queues is the *passive* queue which contains an item for each passive (completed) prefix firing. There is one passive queue for each schedule. Essentially, we again use two layers of priority queues. Whenever a set of previously passive prefix firings, represented as a passive priority queue, is reactivated because of a new prefix extension or because of a schedule merge, this passive priority queue becomes a local priority queue and has a representative inserted into the global priority queue. If such a representative is the highest priority item in the global priority queue, and an `execute`/0 call is made, then the highest priority prefix firing of the represented local queue is removed and dealt with as an ordinary prefix firing. The representatives of local priority queues are updated (and potentially removed) similarly to how this is done in the $\mathcal{W}(r,t)$ data structure of (Ganzinger and McAllester 2002).

Example 5.8 Figure 5.2 illustrates the prefix blocks, the different types of priority queues, and their contents. The global queue, which is shared by all schedules, contains the rule firings RF_1 and RF_2, the prefix firings PF_1, PF_4, PF_5 and PF_8 (the last of which belongs to another schedule), and the local queue representative LQ_1. The represented local queue contains the prefix firings PF_2 and PF_3 which are by definition also in the same prefix block. The schedule's passive queue contains the prefix firings PF_6 and PF_7. The schedule has two prefix blocks, which are associated with respectively the prefix extensions PE_1 and PE_2. □

Using our approach, the cost of deleting items from the global priority queue can be amortized to one of the following events: a new rule firing, a new prefix firing, a new prefix extension (for each representative of a local priority queue), or a match between a prefix firing and a prefix extension (which corresponds to either a new larger prefix firing, or a rule firing).

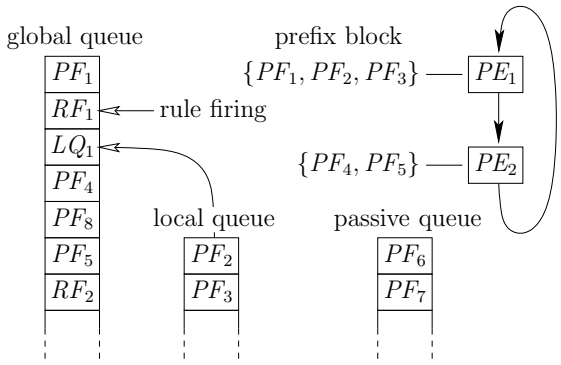


Figure 5.2: Example schedule with global, local and passive priority queues

In (Ganzinger and McAllester 2002), retrieving the schedule for a given prefix firing or prefix extension is done by hashing. In our approach, we use a variant of hashing, which we call *non-ground hashing* and which consists of first replacing all variables by a unique identifier, and then using the resulting (ground) term for hashing. Unifications may require rehashing the affected keys and potentially also the merging of schedules.

5.5.4 Priority Queues

A priority queue or heap is a data structure that contains a set of prioritized items and supports the following operations: inserting and removing an item, finding a highest priority item and merging with another queue. The implementation proposal by Ganzinger and McAllester (2002) suggests the use of two types of priority queues, one for the fixed priorities, where each of the supported operations takes constant time, and Fibonacci heaps for the dynamic priorities.

Fibonacci heaps (Fredman and Tarjan 1987) are a type of priority queue that offer $\mathcal{O}(1)$ amortized time insertion, heap merging and finding a highest priority item, and $\mathcal{O}(\log n)$ amortized time item removal with n the number of items in the queue. It is suggested by Ganzinger and McAllester (2002) that by using only one node per priority, using linked lists to represent the items that share this priority, the item removal cost can be reduced to $\mathcal{O}(\log N)$ with N the number of distinct priorities. However, this increases the cost of heap merging from $\mathcal{O}(1)$ for a single merge operation to a total cost of $\mathcal{O}(n \log N)$ for merging heaps when there are n items in total and N distinct priorities (as is shown in an Appendix of (De Koninck et al. 2007)). A CHR implementation of Fibonacci heaps is described by Sneyers et al. (2006a). It can easily be extended to support multiple heaps that can be merged and to use only one node for each distinct priority per heap.

5.6 A New Meta-Complexity Result for CHR^{rp}

In this section, we give a new meta-complexity result for CHR^{rp}. It extends the result via translation to Logical Algorithms, by also supporting built-in constraints and non-ground CHR constraints. In the following, we assume that hash tables support $\mathcal{O}(1)$ insertion, removal, and retrieval of all elements that match a given (ground) key. This assumption is also made by Ganzinger and McAllester (2002) and holds on average as long as the hash function is good enough. Our scheduling data structure of (De Koninck 2007) makes use of the union-find algorithm, which results in a factor $\alpha(n)$ in the complexity of its operations, where α is the inverse of the Ackermann function. Since this inverse is positive and less than 5 for all practical values of n, we ignore this factor in our complexity result.

We start by looking at the complexity of the different operations supported by our scheduler.

Lemma 4 (Scheduler Costs) *Let N be the number of distinct priorities, and assume that a priority queue merge takes some abstract time T, then the schedule operations have the following amortized cost:*

- $\mathcal{O}(1)$ *and* $\mathcal{O}(\log N)$ *for each* `schedule_pf/3`, `remove_pf/1`, `remove_pe/1`, `remove_rf/1` *and* `execute/0` *operation involving respectively a static and dynamic priority rule*
- $\mathcal{O}(T+1)$ *and* $\mathcal{O}(T+\log N)$ *for each* `schedule_pe/2` *operation involving respectively a static and dynamic priority rule*
- $\mathcal{O}(1)$ *for each schedule merge and* `schedule_rf/2` *operation*

Proof: We only consider the costs related to the priority queue operations. The other costs are shown to be (quasi) constant in (De Koninck 2007). We now look at the different operations in detail:

- A `schedule_pf`/3 call consists of inserting the new prefix firing into the global priority queue. We also account to this event, the cost of making the new prefix instance passive the first time. That operation consists of a removal from the global priority queue and an insertion into the schedule's passive queue. The total cost is $\mathcal{O}(1)$ if the element has a static priority, and $\mathcal{O}(\log N)$ if it has a dynamic priority.
- A `schedule_pe`/2 call requires the insertion of a new representative for the local priority queue of reactivated prefix firings, into the global priority queue. We also take into account here, the cost of making all the reactivated prefix firings passive that have not been matched with a prefix extension since the reactivation. That operation consists of removing the representative and merging the local priority queue with the schedule's passive queue.

The cost is $\mathcal{O}(T+1)$ for a static priority rule and $\mathcal{O}(T+\log N)$ time for a dynamic priority one.

- A `schedule_rf`/2 call requires an insertion into the global priority queue which takes $\mathcal{O}(1)$ time.
- A `remove_pf`/1 call consists of deleting the prefix firing from the global priority queue, from a local priority queue or from a passive queue. A deletion from a local queue may moreover require an update of the global queue (removal and insertion). In total, this takes $\mathcal{O}(1)$ time for a static priority rule and $\mathcal{O}(\log N)$ time for a dynamic priority rule.
- A `remove_pe`/1 call does not require any priority queue operations, and so the cost is $\mathcal{O}(1)$.
- A `remove_rf`/1 call requires a removal from the global priority queue which takes $\mathcal{O}(1)$ time if it involves a static priority rule and $\mathcal{O}(\log N)$ time if it involves a dynamic priority rule.
- An `execute`/0 call requires retrieval and potential removal (if the retrieved item corresponds to a rule firing, or to a prefix firing that becomes passive) of the highest priority item in the global priority queue. If the retrieved item represents a prefix firing or set of prefix firings that need to be made passive, the cost of this operation is already accounted for by a previous `schedule_pf`/3 or `schedule_pe`/2 operation. In such case, we call the `execute`/0 call *unsuccessful*. An unsuccessful `execute`/0 call is followed by another `execute`/0 call until either such a call is successful, or the global priority queue is empty and thus a final state is reached. The cost of all unsuccessful `execute`/0 calls can be amortized to previous events. If in case of a successful `execute`/0 call, the item retrieved from the global priority queue corresponds to the representative of a local priority queue, the operation requires a removal of the highest priority item (prefix firing) from this local queue, an insertion of the prefix firing into the global priority queue, and potentially the insertion of a new representative for the local queue into the global queue. The cost of a successful `execute`/0 call therefore equals $\mathcal{O}(1)$ if it involves a static priority rule and $\mathcal{O}(\log N)$ otherwise.
- A schedule merge requires the reactivation of the passive prefix firings of the merged schedules. The cost analysis is similar to that of a `schedule_pe`/2 call. Moreover, each schedule merge can be accounted for by at least one `schedule_pf`/3 or `schedule_pe`/2 call as the resulting schedule contains at least one prefix firing or extension more than each of the original schedules, and so the number of schedule merges is bounded by the number of prefix firings and extensions. Therefore, the cost of a single schedule merge can be considered constant.

□

In the above lemma, we have made abstraction of the cost of priority queue merge operations. Such merges take place when the prefix firings in a local priority queue all become passive. In such an event, the local priority queue is merged with the schedule's passive queue. It is easy to see that the cost of merging priority queues for static priorities takes constant time per merge operation. In Section 5.5 a bound is given on the total cost of merging Fibonacci heaps with one node per distinct priority, given the number of items ever inserted into the heaps. The following lemma makes use of this result.

Lemma 5 (Fibonacci Heap Merging Cost) *The total cost of Fibonacci heap merges is* $\mathcal{O}((P_d + A_d) \cdot \log N)$ *where* P_d *is the number of strong prefix firings of dynamic priority rules,* A_d *is the number of constraints that may participate in a dynamic priority rule instance, and* N *is the number of distinct rule priorities.*

Proof: We count the number of items ever inserted into the local and passive Fibonacci heaps, and then apply the result of Section 5.5. A local priority queue basically is the same as a passive priority queue in which items are no longer inserted. Therefore, a merge between a local queue and a passive queue can be seen as a special case of a merge between two passive queues and so we only need to consider these passive priority queues. Each item inserted in such a queue is either a prefix firing that has never been passive before, or a prefix firing that has been matched with a prefix extension at least once since its last activation. The total number of these items is $\mathcal{O}(P_\mathsf{d} + A_\mathsf{d})$ because each prefix firing that has been matched with a prefix extension is by definition a strong prefix firing, and each new prefix firing either results from matching a (smaller) strong prefix firing and extension and hence corresponds to a (potentially suspended) strong prefix firing, or consists of a single head in which case it corresponds to a constraint assertion. Now given the number of items ever inserted into the passive priority queues, the total cost of merging Fibonacci heaps hence is $\mathcal{O}((P_\mathsf{d} + A_\mathsf{d}) \cdot \log N))$. □

We are now ready to formulate the new meta-complexity theorem.

Theorem 11 *Let* A_s *and* A_d *be the number of assertions of constraints with an occurrence in respectively a static and dynamic priority rule. Let* P_s *and* P_d *be the number of strong prefix firings of respectively static and dynamic priority rules. The time complexity of a CHR^{rp} program executed using our implementation is*

$$\mathcal{O}(C_{ask} \cdot (A_s + P_s + (A_d + P_d) \cdot \log N) + B \cdot C_{tell} \cdot (K + C_{ask} \cdot S))$$

where N *is the number of distinct priorities,* C_{ask} *is the cost of evaluating a built-in ask constraint,* C_{tell} *is the cost of solving a built-in tell constraint, and* B *is the number of built-in tell constraints asserted in rule bodies;* K *is the maximum number of distinct combinations (keys) of arguments shared between prefix firings and extensions in which any given variable occurs, and* S *is the maximum number*

of suspended strong prefix firings (i.e., those that are followed by a non-trivial guard) and suspended instances of constraint occurrences (i.e., whose arguments are not mutually distinct variables) in which any given variable occurs.

Proof: Each new CHR constraint causes the creation of constraint occurrences which are converted into RETE memory constraints as soon as the implicit guard on the constraint arguments is entailed (i.e., the constraint matches the head in question). These RETE memory constraints are scheduled using `schedule_pf`/3 for the single-headed prefix firings, `schedule_rf`/2 for the single-headed rule firings, and `schedule_pe`/2 for the prefix extensions. The total cost of these operations, including the cost of priority queue merges (for the `schedule_pe`/2 calls), equals $\mathcal{O}(C_{\mathsf{ask}} \cdot (A_{\mathsf{s}} + (A_{\mathsf{d}} + P_{\mathsf{d}}) \log N))$. Each constraint deletion causes the deletion of those RETE memory constraints in which the deleted constraint participated. The total cost related to deletion therefore is $\mathcal{O}(A_{\mathsf{s}} + P_{\mathsf{s}} + (A_{\mathsf{d}} + P_{\mathsf{d}}) \log N)$. Each prefix firing is inserted into its schedule at most once and hence it can also be removed from this schedule only once (when one of its constituent constraints is removed). Those prefix firings that consist of at least two heads, correspond to a strong prefix firing as they are generated at a priority higher than or equal to that of the highest priority rule firing. Thus, using Lemma 4 and including the cost of checking the relevant parts of the guard, the cost for inserting (and deleting) these prefix firings is $\mathcal{O}(C_{\mathsf{ask}} \cdot (P_{\mathsf{s}} + P_{\mathsf{d}} \log N))$.

A built-in tell constraint is processed as follows. The keys used to identify the schedules and that are affected by the built-in constraint, are rehashed. If the built-in constraint causes two or more schedules to have the same key, these schedules are merged. The cost of rehashing is proportional to the number of affected keys and the cost of a schedule merge is constant by Lemma 4. A built-in constraint moreover requires the reactivation of the suspended prefix firings and rule firings, as well as those constraint occurrences for which it is not decided whether they match with the corresponding head or not. The reactivated prefix and rule firings have their guard checked and are potentially scheduled as regular (non-suspended) prefix and rule firings. The reactivated constraint occurrences also have their (implicit) guard checked, and are potentially scheduled as single-headed prefix firings, single-headed rule firings, or prefix extensions. The cost of the scheduling operations was already taken into account above. The remaining cost per built-in tell constraint is $\mathcal{O}(C_{\mathsf{tell}} \cdot (K + C_{\mathsf{ask}} \cdot S))$. □

Because the values of S and K might be difficult to determine in practice, we propose the following bounds: $S = \mathcal{O}(A_{\mathsf{s}} + A_{\mathsf{d}} + P_{\mathsf{s}} + P_{\mathsf{d}})$ and $K = \mathcal{O}(A_{\mathsf{s}} + A_{\mathsf{d}})$. We have used the cost of solving a built-in tell constraint as an upper-bound on the number of variables that are affected. Note that in absence of built-in constraints, the theorem given here is essentially the same as the one for Logical Algorithms.[10]

[10]The Logical Algorithms result makes use of the size of the initial database instead of the number of assertions. We have that $A_{\mathsf{s}} = \mathcal{O}(|\sigma_0| + P_{\mathsf{s}} + P_{\mathsf{d}})$.

5.6.1 Examples

We illustrate the meta-complexity theorem on some examples, and compare with the results obtained by using the approach of Frühwirth (2002b).

Example 5.9 (Less-or-Equal) A first example is the `leq` program, given in regular CHR by Listing 2.1 and in CHR^{rp} by Listing 3.1. We use a slightly different priority assignment compared to the version of Listing 3.1 to simplify the analysis. In particular, we have given the `idempotence` rule a higher priority.

```
1 :: idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
2 :: reflexivity  @ leq(X,X) <=> true.
2 :: antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
3 :: transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Given an initial goal consisting of n `leq`/2 constraints where the arguments are taken from a set of n distinct variables, we derive the following values for the parameters:

- P_s: the number of strong prefix firings is $\mathcal{O}(n^2)$ for the `idempotence` rule, $\mathcal{O}(n)$ for the `reflexivity` rule, $\mathcal{O}(n^2)$ for the `antisymmetry` rule, and $\mathcal{O}(n^3)$ for the `transitivity` rule. These numbers are found by looking at the degrees of freedom for each constraint occurrence, based on the domain of the arguments, and given those arguments that are already fixed by the left-most heads. For example for the `transitivity` rule, we know that there are $\mathcal{O}(n^2)$ constraints matching the first head, and $\mathcal{O}(n)$ constraints matching the second head, given the first. Our reasoning is based on the fact that at priority 2 and lower, all `leq`/2 constraints have set semantics because of the `idempotence` rule.[11]
- A_s: the number of `leq`/2 constraints asserted is $\mathcal{O}(n^3)$ (by the `transitivity` rule).
- B: the number of built-in constraints is bounded by the number of rule firings of the `antisymmetry` rule, and hence is $\mathcal{O}(n^2)$.
- K: the schedule keys are the combination of `X` and `Y` in both the `antisymmetry` rule and the `idempotence` rule, and `Y` in the `transitivity` rule. There are at most $\mathcal{O}(n)$ different keys in which any given variable occurs.
- S: for any variable, and in a state in which a built-in constraint can be asserted, there are up to $\mathcal{O}(n)$ suspended instances of the `leq`/2 occurrence in the `reflexivity` rule. There can be no suspended prefix or rule firings.

[11] We assume here that the `idempotence` rule always removes the most recently asserted duplicate, see Section 3.5 for a way to ensure this in CHR^{rp}.

- C_{ask} and C_{tell}: the cost of evaluating a built-in ask constraint and the cost of solving a built-in tell constraint is constant (at least for the given query pattern).

Filling in these parameters in the formula given by Theorem 11 gives us a worst case time complexity of

$$\mathcal{O}(1 \cdot (n^2 + n^3) + (0 + 0) \cdot \log 1) + n^2 \cdot 1 \cdot (n + 1 \cdot n)) = \mathcal{O}(n^3)$$

This corresponds to the actual worst-case complexity for an initial goal of the form

$$\{\texttt{leq}(X_1, X_2), \ldots, \texttt{leq}(X_{n-1}, X_n), \texttt{leq}(X_n, X_1)\}$$

The approach of Frühwirth (2002b) does not apply since the `transitivity` rule is a propagation rule and hence no suitable ranking function can be found. □

Example 5.10 (Merge Sort) Consider the CHR$^{\text{rp}}$ implementation of the merge sort algorithm, first given in Example 5.6 (Section 5.4) and repeated here for easy reference.

```
1 :: ms1 @ arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).
2 :: ms2 @ merge(N,A),  merge(N,B) <=> A < B | merge(2*N+1,A), arrow(A,B).
3 :: ms3 @ number(X) <=> merge(0,X).
```

We show that the total runtime of the algorithm is $\mathcal{O}(n \log n)$ given an initial goal consisting of n `number`/1 constraints.

No new `number`/1 constraints are ever asserted. Rule `ms3` converts one `number`/1 constraint into one `merge`/2 constraint each time it fires. The number of (strong) prefix firings for rule `ms3` hence is $\mathcal{O}(n)$. Rule `ms2` decreases the number of `merge`/2 constraints by one and so it can fire $n - 1$ times. In any state, there are at most two `merge`/2 constraints with the same first argument. This invariant holds in the initial state because there are no `merge`/2 constraints in the initial goal and rule `ms2` can fire after each new `merge`/2 constraint assertion, enforcing the invariant. Because of the invariant, the number of prefix firings for rule `ms2` is limited to $\mathcal{O}(n)$.

Using similar reasoning it holds that in any state, there are at most two `arrow`/2 constraints in the store with the same first argument. Now we define that in a given state, two numbers x_1 and x_m are connected by a *chain* of length $m - 1$ if the constraint store contains $\texttt{arrow}(x_1, x_2), \texttt{arrow}(x_2, x_3), \ldots, \texttt{arrow}(x_{m-1}, x_m)$. At priority 2 it holds that for each $\texttt{merge}(m, x)$ constraint in the store, the maximal length of a chain starting in x is m. Indeed, this holds for the initial $\texttt{merge}(0, \_)$ constraints and if it holds for $\texttt{merge}(m, \_)$ constraints, it also holds for $\texttt{merge}(2 \cdot m + 1, \_)$ constraints, because when such a constraint is asserted, two chains of length m are linked with an extra `arrow`/2 constraint and merged by up to $2 \cdot m$ firings of rule `ms1`. Two $\texttt{merge}(m, \_)$ constraints are combined into a $\texttt{merge}(2 \cdot m + 1, \_)$

constraint, so the n `merge`$(0, \_)$ constraints asserted by rule `ms3` are replaced by $n/2$ `merge`$(1, \_)$ constraints, which in turn are combined into $n/4$ `merge`$(3, \_)$ constraints and so on until finally 1 `merge`$(n-1, \_)$ constraint remains. The sum of all m in these `merge`$(m, \_)$ constraints is $\mathcal{O}(n \log n)$. Rule `ms1` fires $\mathcal{O}(m)$ times after every new `merge`$(m, \_)$ constraint assertion and because there are at most two `arrow`/3 constraints with the same first argument, there are $\mathcal{O}(n \log n)$ strong prefix firings of rule `ms1`.

In conclusion, for an input database consisting of n `number`/1 constraints, there are $\mathcal{O}(n \log n)$ strong prefix firings for rule `ms1`, $\mathcal{O}(n)$ for rule `ms2` and $\mathcal{O}(n)$ for rule `ms3`. Using the meta-complexity theorem, which simplifies to the one for Logical Algorithms because there are no built-in tell constraints, the total runtime is $\mathcal{O}(n \log n)$, which is also a tight complexity bound. We now compare this result with the result found by using the meta-complexity theorem of Frühwirth (2002b).

Using a similar analysis as above, we can derive that $D = \mathcal{O}(n \log n)$ and $c_{\mathsf{max}} = \mathcal{O}(n)$ where n is the number of `number`/1 constraints in the query.[12] The cost of head matching (O_{H_r}), guard checking (O_{G_r}), adding built-in constraints (O_{C_r}), and adding and removing CHR constraints (O_{B_r}), can all be assumed constant. The number of heads n_r of a rule $r \in P$ is at most 2. Filling in these numbers, we derive a total worst case complexity of $\mathcal{O}(n^3 \log n)$, which is clearly suboptimal. □

5.6.2 Comparison with the LA Meta-Complexity Result

In (De Koninck et al. 2007), we have presented a direct implementation of the Logical Algorithms language into CHR that satisfies the complexity requirements needed for the Logical Algorithms meta-complexity result to hold. In this subsection, we show that this implementation has become somewhat obsolete because we can achieve the same result by combining the translation from Logical Algorithms to CHR^{rp} of Section 5.3, with the CHR^{rp} implementation presented in Section 5.5. We assume here that the comparison antecedents in Logical Algorithms programs are scheduled after the corresponding user-defined antecedents in the translation, and that the guards on the mode indicators (these have the form $N \neq \mathtt{p}$) are scheduled right after the head to which they apply.

Theorem 12 *The time complexity of Logical Algorithms programs executed by first translating them into CHR^{rp} programs using the translation schema of Section 5.3, and then executing the resulting CHR^{rp} program using the implementation of Section 5.5, is $\mathcal{O}(|\sigma_0| + P_s + (P_d + A_d) \cdot \log N)$ with σ_0, P_s, P_d, A_d and N as defined in Section 5.2.1.*

[12] In Theorem 4.2 of (Frühwirth 2002b) a worst case upper-bound of $c_{\mathsf{max}} = \mathcal{O}(c + D)$ is used, with c the number of constraints in the query, which becomes $c_{\mathsf{max}} = \mathcal{O}(n \log n)$ in this example. The bound we use is tight, i.e., $c_{\mathsf{max}} = \Theta(n)$.

Proof: The translation of a Logical Algorithms program P consists of two parts as defined in Section 5.3. The first part, denoted by $T_{S+D}(P)$, contains for each user-defined predicate a/n the following rules:

$$\begin{aligned} 1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus a(\bar{X}) &\iff M \neq \mathtt{n} \mid \mathtt{true} \\ 1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{n}), a(\bar{X}) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \\ 2 :: a(\bar{X}) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{p}) \end{aligned}$$

$$\begin{aligned} 1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus \mathtt{del}(a(\bar{X})) &\iff M \neq \mathtt{p} \mid \mathtt{true} \\ 1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{p}), \mathtt{del}(a(\bar{X})) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \\ 2 :: \mathtt{del}(a(\bar{X})) &\iff a_{\mathtt{r}}(\bar{X}, \mathtt{n}) \end{aligned}$$

It is easy to see that for an initial goal containing no constraints of the form $a_{\mathtt{r}}(\bar{X}, M)$ and since these are the only rules that assert such a constraint, in any state it holds that if $a_{\mathtt{r}}(\bar{X}, M_1)\#i_1$ and $a_{\mathtt{r}}(\bar{X}, M_2)\#i_2$ are in the CHR constraint store, then $i_1 = i_2$ and $M_1 = M_2$. This implies that the number of strong prefix firings for these rules is bounded by the number of assertions of $a(\bar{X})$ or $\mathtt{del}(a(\bar{X}))$.

The second part of the translation, denoted by $T_R(P)$, contains for each Logical Algorithms rule

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

a set of rules

$$p + 2 :: r_\rho \ @ \ H \implies g_1, g_2 \mid C'$$

as shown in the translation schema of Section 5.3.1. Amongst these rules is one, say $r_{\rho'}$, with a maximal number of heads, namely as many as there are user-defined antecedents in $A_1, \ldots, A_n$. Because the (implicit and explicit) guards on the mode indicators of the head constraints are scheduled as soon as they are decidable, and because the comparisons are scheduled at corresponding places, it is easy to see that the number of strong prefix firings of rule $r_{\rho'}$ is the same as the number of strong prefix firings of Logical Algorithms rule r. The other r_ρ rules are restricted versions of $r_{\rho'}$ and therefore have at most as many strong prefix firings as $r_{\rho'}$.

The assertions with occurrences in dynamic priority rules are of the form $a_{\mathtt{r}}(\bar{X}, \_)$. The set and deletion semantics rules ensure that the number of these assertions is the same in the original program and in its translation. Now using our new meta-complexity result for CHR$^{\mathrm{rp}}$ (Theorem 11), we derive that the total runtime complexity of the translated program is $\mathcal{O}(|\sigma_0| + P_{\mathtt{s}} + (P_{\mathtt{d}} + A_{\mathtt{d}}) \cdot \log N)$.[13] □

5.6.3 Comparison with the "As Time Goes By" Approach

In Section 5.2.2 we already briefly compared the LA meta-complexity result with the theorem given by Frühwirth (2002b). In this subsection, we make the com-

[13] Since $A_{\mathtt{s}} = \mathcal{O}(|\sigma_0| + P_{\mathtt{s}} + P_{\mathtt{d}})$.

parison complete by also considering built-in constraints, using the new meta-complexity theorem presented in Section 5.6.

Let there be given a CHR$^{\text{rp}}$ program P in which each rule has the same (static) priority. Theorem 3 in Section 3.3.3 states that such a CHR$^{\text{rp}}$ program and its corresponding CHR program (which is found by removing the rule priorities) have the same derivations. Therefore, such programs are suitable for comparing the result of (Frühwirth 2002b) with the result of Theorem 11 in Section 5.6. In Section 5.2.2 we have already shown that the number of strong prefix firings is $\mathcal{O}\left(D \cdot \sum_{r \in P} c_{\mathsf{max}}^{n_r}\right)$ where D is the derivation length (i.e., the number of rule firings), and c_{max} is the maximal number of CHR constraints in the store in any state. The number of constraint assertions is $\mathcal{O}(c_{\mathsf{max}} + D)$. If we assume that the initial goal does not contain any built-in constraints (as is done in (Frühwirth 2002b)), then the number of built-in constraints is $\mathcal{O}(D)$. The number of suspended prefix firings is bounded by $\mathcal{O}\left(\sum_{r \in P} c_{\mathsf{max}}^{n_r}\right)$ in any state and the number of suspended assertions by $\mathcal{O}(c_{\mathsf{max}})$. Now, filling in these parameters in the CHR$^{\text{rp}}$ meta-complexity result gives us that the total runtime complexity is

$$\mathcal{O}\left(O_C \cdot D \sum_{r \in P} (c_{max}^{n_r} \cdot O_{G_r})\right) \tag{5.3}$$

where $O_C = \sum_{r \in P}(O_{C_r})$. This formula strongly resembles the result of Frühwirth (2002b) which, assuming the cost of head matching O_{H_r} and adding and removing CHR constraints O_{B_r} is constant, equals

$$\mathcal{O}\left(D \sum_{r \in P} (c_{max}^{n_r} \cdot O_{G_r} + O_{C_r})\right) \tag{5.4}$$

The difference lies in how built-in tell constraints are dealt with. In our CHR$^{\text{rp}}$ implementation, as well as in any CHR implementation based on the refined operational semantics of CHR, a built-in tell constraint causes the constraints or matches whose variables are affected, to be reconsidered.[14] Because each individual (atomic) built-in constraint is dealt with separately, this may cost more in total than the naive approach taken in (Frühwirth 2002b) in which after each rule firing, *all* constraints or matches are reconsidered *once*. So, while in certain rather exceptional cases, a naive approach to dealing with built-in tell constraints might in fact be better than the usual approach of selective reactivation (as can be seen by comparing Formulas (5.3) and (5.4)), in general we expect the latter approach to be an improvement over the naive one. Moreover, in these exceptional cases, the meta-complexity theorem of (Frühwirth 2002b) does not apply to optimized CHR implementations such as the K.U.Leuven CHR system, i.e., in these cases it does not overestimate the actual worst case time complexity.

[14] Which constraints are reactivated depends on the wake-up policy used for the **Solve** transition, see also (Schrijvers 2005, Section 5.4.2).

5.7 Conclusions

In this chapter, we have investigated the relationship between the Logical Algorithms language and Constraint Handling Rules. We have presented an elegant translation schema from Logical Algorithms to CHR^{rp}. The original program and its translation are shown to be essentially weakly bisimilar. However, our current CHR^{rp} system (see Chapter 3) does not give the complexity guarantees needed for the Logical Algorithms meta-complexity theorem to hold via this translation.

As a first step towards applying the Logical Algorithms meta-complexity result to CHR^{rp} programs, we have shown how a subclass of CHR^{rp} can be translated into Logical Algorithms. By using this translation, we can directly apply the meta-complexity theorem for Logical Algorithms to the translated CHR^{rp} programs. A drawback is that the CHR^{rp} programs that can be translated this way, are restricted to those that do not make use of an underlying constraint solver.

In order to remedy both the limitation that the translation from Logical Algorithms to CHR^{rp} does not exhibit the required complexity when executing translated Logical Algorithms programs using our CHR^{rp} system, and the restriction of those CHR^{rp} programs that can be translated to Logical Algorithms and hence to which the Logical Algorithms meta-complexity result can be applied, we have proposed a new implementation for the complete CHR^{rp} language that gives strong complexity guarantees. The implementation is based on the high-level implementation proposal of Ganzinger and McAllester (2002) as well as on the scheduling data structure of De Koninck (2007), and consists of the compilation of CHR^{rp} rules into (regular) CHR rules, combined with a scheduler that controls the execution. The implementation supports a new and accurate meta-complexity theorem for CHR^{rp}. When combining the translation from Logical Algorithms to CHR^{rp} with the new implementation, the new meta-complexity theorem implies the Logical Algorithms meta-complexity result. Moreover, it is shown that in general[15] the new theorem is at least as accurate as the meta-complexity result for CHR given by Frühwirth (2002b). This is illustrated on two non-trivial examples, one of which contains both built-in constraints and propagation rules and therefore cannot be analyzed using the Logical Algorithms approach or Frühwirth's result.

5.7.1 Related Work

The time complexity of programs is in general expressed in terms of the number of elementary operations, e.g., the number of logical inferences in Prolog, function applications in a functional programming language, or rule applications in a language such as CHR. However, while in most languages, these elementary operations all take constant time,[16] this is not the case in a language like CHR where

[15] Apart from some rather exceptional cases, see Section 5.6.3.

[16] Prolog unification takes more than constant time in general, but not under certain restrictions such as those imposed in Mercury (Somogyi et al. 1996).

each rule application results from a complex matching phase.

In this work, we have made a mapping from the number of elementary operations (like prefix and rule firings or constraint assertions) to time complexity. To the best of our knowledge, and apart from the results in (McAllester 1999; Ganzinger and McAllester 2001; Ganzinger and McAllester 2002) and (Frühwirth 2002a; Frühwirth 2002b), there is no other work with a similar goal. There are many other formalisms though in which elementary operations take more than constant time. One such formalism is term rewriting, as implemented by the Maude system (Clavel et al. 1999) or the ACD term rewriting language (Duck et al. 2006). It is known that AC matching, which is used by most of these languages, is NP-complete. Another formalism is that of production rule systems like Drools (Proctor et al. 2007) or Jess (Friedman-Hill 2007). Production rules are in many ways similar to Constraint Handling Rules. However unlike CHR, these systems are not often used as general purpose programming language, and therefore, algorithmic complexity has never been much of a concern. More work exists on the derivation of the number of elementary operations. In the context of CHR, this mostly concerns the number of rule firings, which is often derived as part of termination analysis (Frühwirth 2000; Pilozzi et al. 2007; Voets et al. 2007).

Another related topic is that of space complexity, an issue that is not dealt with in this thesis. In the context of CHR, the memory reuse techniques developed by Sneyers et al. (2006b) are crucial to achieve optimal space complexity as is shown in (Sneyers et al. 2008). The latter also introduces a space complexity meta-theorem for CHR, stating that the space complexity is $\mathcal{O}(D + p)$ where D is the derivation length and p is the number of propagation rule firings (which takes into account the size of the propagation history).

5.7.2 Future Work

In a previous version of this work (De Koninck et al. 2007), we have made an actual implementation for the Logical Algorithms language in CHR. This implementation satisfies the complexity requirements needed for the Logical Algorithms meta-complexity theorem to hold, when executed using the K.U.Leuven CHR system on top of SWI-Prolog. However, the very large constant factors and the high memory consumption makes the system not very useful in practice. Currently, we have no running version of the alternative implementation for CHR^{rp} presented in Section 5.5. In general, it would be interesting to evaluate the advantages and disadvantages of a lazy RETE based matching algorithm for CHR$^{(rp)}$ compared to the LEAPS style matching that is currently used by almost all systems.

We have already mentioned in the related work discussion that a space complexity result for our alternative implementation is currently lacking. The RETE style matching we used is in general far from optimal as far as memory usage is concerned, in particular compared to LEAPS style matching as is used by most

CHR systems. However, in the CHR context, built-in constraints may require maintaining a propagation history which in the worst case requires as much memory as the alpha and beta memories in RETE matching. Therefore, it would be interesting to more formally compare both styles of matching in terms of memory consumption in the context of CHR.

Chapter 6

Join Ordering for CHR

Join ordering is the problem of finding cost optimal execution plans for matching multi-headed rules. In the context of Constraint Handling Rules, this topic has received limited attention so far, even though it is of great importance for efficient CHR execution. We present a formal cost model for joins and investigate the possibility of join order optimization at runtime. We propose some heuristic approximations of the parameters of this cost model, for both the static and dynamic case. We discuss an $\mathcal{O}(n \log n)$ optimization algorithm for the special case of acyclic join graphs. However, in general, join order optimization is an NP-complete problem. Finally, we identify some classes of cyclic join graphs that can be reduced to acyclic ones.

6.1 Introduction

Much work has been devoted to the optimized compilation of CHR (Duck 2005; Holzbaur et al. 2005; Schrijvers 2005). A crucial aspect of CHR compilation is finding matching rules efficiently. Given an active constraint, searching for matching partner constraints corresponds to joining relations — a well-studied topic in the context of databases (Bayardo and Miranker 1996; Krishnamurthy et al. 1986; Selinger et al. 1979; Steinbrunn et al. 1997; Swami and Iyer 1993). The performance of join methods depends on indexing and join ordering.

In the context of CHR, join ordering has been discussed in (Duck 2005; Holzbaur et al. 2005). In these works, only static (compile-time) information is used in determining an optimal join order. Moreover, little attention is given to the complexity of the optimization algorithm, since join ordering is done at compile time and because the input sizes (the number of heads of a rule) are expected to be very small ("usually at most 3 in practice" (Duck 2005)). In this chapter we consider dynamic (run-time) join ordering based on constraint store statistics such as selectivities

and cardinalities. We also take into account that CHR programs increasingly contain rules with more than three heads. (Some examples of such programs are given in Section 6.7.2.)

The main contributions of this chapter are the following. We formulate a generic cost model for join ordering, which provides a solid foundation to develop and evaluate heuristics. We introduce dynamic join ordering. Finally we discuss algorithms to optimize the join order that were first introduced in the database context. In particular, an efficient $\mathcal{O}(n \log n)$ algorithm for rules with acyclic join graphs is presented.

Example 6.1 Consider the following rule from a CHR implementation[1] of an algorithm by Hopcroft (1971) for minimizing states in a finite automaton:

```
partition(A,I), delta(T,A,X), a(A,I,X) \ b(J,T) <=> b_prime(J,T).
```

In this rule, when `partition(A,I)` is activated, in order to achieve the correct time complexity, `A` and `I` should be used to find matching `a(A,I,X)` constraints; then `A` and `X` can be used to find matching `delta(T,A,X)` constraints; the resulting `T` can finally be used to find a matching `b(J,T)` constraint. However, the K.U.Leuven CHR system currently uses `A` to find `delta(T,A,X)`, then `T` to find `b(J,T)`, and finally `A`, `I` and `X` to find `a(A,I,X)`. As a result, the optimal time complexity is not obtained. The heuristic introduced in (Holzbaur et al. 2005) also results in a suboptimal join order if the functional dependencies $\texttt{delta}(t, a, x) :: \{t, a\} \rightsquigarrow \{x\}$ and $\texttt{b}(j, t) :: \{t\} \rightsquigarrow \{j\}$ are taken into account.[2] This rule and many other rules for which current systems use a suboptimal join order motivate a closer investigation of the join ordering problem for CHR. □

The rest of this chapter is organized as follows. Section 6.2 gives a high level discussion of our approach. In Section 6.3, we give a generic cost model for computing a join, whose parameters are estimated in Section 6.4. Section 6.5 extends our cost model towards CHR^{rp} by taking into account the cost of scheduling items in the agenda. Next, in Section 6.6, alternative methods for computing an optimal join order are investigated. Section 6.7 discusses some remaining issues and we conclude in Section 6.8.

6.2 Basic Approach

6.2.1 Processing Trees

A join execution plan can be represented by a binary tree, called the processing tree. The leaves of a processing tree represent the different heads to be joined

[1] All example programs are available from (De Koninck and Sneyers 2008).

[2] There is also a functional dependency $\texttt{a}(a, i, x) :: \{x\} \rightsquigarrow \{a, i\}$.

and its nodes represent the (partial) join of two or more heads. One particularly interesting type of processing trees are the so-called *left-deep* trees which are used by most database systems, as well as current CHR implementations. A left-deep processing tree is inductively defined as follows: a single head is a left-deep tree, and a tree in which the root node has a left-deep tree as its left child and a head as its right child is also a left-deep processing tree.

A join proceeds by traversing all tuples of the left child of the join node and looking up appropriate tuples of the right child for each of them. This is often called a *nested loop* join. It supports pipelining which means that intermediate results do not need to be stored. In this chapter, we assume that partial joins are not indexed, i.e., if they are temporarily stored, we can only traverse them by sequential search. This implies that a *right-deep* processing tree[3] is problematic, since we have no indexing support to perform the nested loop joins.

The most general type of processing trees are called *bushy trees*. Both the left and right child of a bushy tree node can be a (non-trivial) processing tree itself. In general, bushy trees do not support pipelining and also using indexes is often a problem. However, there are cases in which these disadvantages are outweighed by reduced intermediate result sizes.

Example 6.2 Consider the rule

```
a(X), b(X,Y), c(Y,Z), d(Z,A), e(A) <=> ...
```

Assume the first head is the active constraint, there are many combinations of the second and third head resulting in the same value for `Z` and there are many `d/2` constraints for any value of its first argument, though few for which a matching `e/1` constraint exists. Clearly, we can waste a lot of effort recomputing the join of `d(Z,A)` and `e(A)` for each combination of `b(X,Y)` and `c(Y,Z)`. In this case it might be better to compute the join of `d(Z,A)` and `e(A)` once and store it for later retrieval. □

In the remainder of this chapter, we only consider left-deep processing trees.

6.2.2 Indexing

When joining a partial match with a head, we can use indexes on constraint arguments for faster retrieval. When no indexes are available, we are forced to consider each instance of a constraint in the store. Modern CHR implementations employ various indexing schemes. For example, the K.U.Leuven CHR system (Schrijvers and Demoen 2004), which is the de facto standard implementation of CHR for Prolog, offers the following indexing structures:

[3] A right-deep processing tree is similar to a left-deep one, but with the left and right child switched.

- Hash table lookup for ground argument positions
- Array lookup for dense integer arguments (the same complexity as hash tables, but lower constant factors)
- Attributed variables indexing: allows quick lookup of all constraints containing a particular variable

Our CHR$^{\text{rp}}$ implementation (Chapter 3) only uses hash tables, but supports the hashing of non-ground terms by first replacing its variables by unique identifiers (*non-ground hashing*). This way, we can support constraint insertion and deletion in $\mathcal{O}(1)$ amortized average-case time, $\mathcal{O}(1)$ lookup of all constraints with a given key, and $\mathcal{O}(n)$ update cost after unification where n is the number of affected constraints. The indexing structure allows for constant time lookup, regardless of the instantiation of the variables. In the remainder of this chapter, we assume non-ground hashing is used for indexing all indexed argument positions. Note that our indexes do not support lookups via partially instantiated *arguments*, e.g.

```
a(X,Y), b(f(_,X,_),Y) ==> ...
```

We refer to (Sarna-Starosta and Schrijvers 2008) for ways to make such lookups more efficient.

Other types of indexes could be proposed, for example search trees for inequality guards ($<$, $>$, $\leq$, $\geq$). Also other often used (explicit) equality guards could be supported, for example for patterns like

```
a(X,Y), b(X,Y1) <=> Y1 =:= Y - 1 | ...
```

We do not consider such indexing structures as they are currently not supported by CHR implementations. Therefore, for guards other than implicit equality guards, we filter out non-matching heads *after* retrieval.

6.2.3 Static and Dynamic Join Ordering

Current CHR implementations attempt to determine an optimal join order (if they optimize at all) at compile time, after analyses such as functional dependency analysis (Duck and Schrijvers 2005). The join order therefore remains fixed during execution. We call this *static* join ordering. Next to static join ordering, we also consider join ordering at runtime: *dynamic* join ordering. In the dynamic case, we can use runtime information such as constraint cardinalities and selectivities. However, at runtime we cannot afford to spend as much time on optimization as at compile time.

Example 6.3 To show the advantages of dynamic join ordering, consider the rule:

Case	C_{Θ_1}	C_{Θ_2}
1	$\sqrt{n}+\sqrt{n}$	$n+\sqrt{n}$
2	$\sqrt{n}+\sqrt{c}$	$c+\sqrt{c}$

Table 6.1: Different optimal join orders under different dynamic conditions

```
supports(A,chelsea), supports(B,liverpool) \ friend(A,B) <=> true.
```

Let there be n `supports`/2 constraints, $\sqrt{n}$ of which have `liverpool` as their second argument. We say that the cardinality of the `supports`/2 constraint is n. Let there be a functional dependency from the first argument of `supports`/2 to the second. We now consider two cases. In the first case, there are n `friend`/2 constraints for every value of the first argument. In the second case, there are only a small constant number (c) of `friend`/2 constraints for each such value. We say that the selectivity of the `friend`/2 constraint given its first argument is n respectively c. Each person initially has a representative number of friends supporting Liverpool, i.e., $\sqrt{n}$ in the first case and $\sqrt{c}$ in the second case.

Let the first head be active. We try two join orders: in Θ_1, the second head is retrieved before the third, and in Θ_2, the third head is retrieved before the second. Table 6.1 shows the costs for the two join orders in each case, using the cost formula of Section 6.3. Clearly, the wrong join order leads to a suboptimal complexity: $\Theta(n)$ versus $\Theta(\sqrt{n})$ in the first case and $\Theta(\sqrt{n})$ versus $\Theta(1)$ in the second. Moreover, the optimal join order depends on runtime statistics that could change during execution, so no static order can be optimal. □

Dynamic join ordering still allows for a broad range of design choices. A join order should be available at the moment an active occurrence starts looking for partner constraints. However, after firing a first rule instance, runtime statistics may change and a different join order may become optimal. If we change the join order in between rule firings for the same active occurrence, we may no longer be able to use the partial joins that have already been computed, which probably outweighs the advantage of a better join order. Another issue is whether an optimal join order should be recomputed each time an occurrence becomes active. Clearly, a previously optimal join order remains optimal as long as runtime statistics do not considerably change, and so we can store it for later reuse.

6.3 Cost Model

We now introduce a cost model to estimate the join cost of a particular join order. Our cost model assumes that rules are propagation rules with bodies that do not fundamentally change the relevant statistics, similar to what is assumed

in the context of join order optimization for database systems. In the case of simplification or simpagation rules, it seems advantageous to optimize for finding the first full matches. In (Bayardo and Miranker 1996), this topic is handled by taking into account the probability that a given partial match cannot be extended into a full match. We return to this issue in Section 6.7.1.

6.3.1 Notation

Consider given an active identified constraint a, trying occurrence H_i $(1 \leq i \leq n)$ of a rule with n heads: $H_1, \ldots, H_n$. A join order Θ for this occurrence is a permutation of $\{1, \ldots, n\}$ such that $\Theta(1) = i$. The matching is done by nested partner lookups in the order $\Theta(2), \Theta(3), \ldots, \Theta(n)$. We also write $\Theta = [\Theta(1), \ldots, \Theta(n)]$, and $|\Theta| = n$, i.e., $|\Theta|$ is the size of the *domain* of Θ. We use the notation $\mathsf{st}(H_i)$ to denote all constraints (with identifier) in the CHR store which have the same constraint predicate as H_i. We use $\bar{h}$ to denote a tuple and h_i to denote its i^{th} element (starting from 1). Suppose the rule has a guard G, which determines which elements of $\mathsf{st}(H_1) \times \ldots \times \mathsf{st}(H_n)$ are part of the join (we consider both the implicit and the explicit guard as part of G). Without loss of generality, this guard can be considered as a conjunction $g_1 \wedge \ldots \wedge g_m$, where each conjunct g_j is a host-language constraint on a subset of the head variables. We define the guard scheduling index $\mathsf{gs\_index}(g_j)$ to be the earliest possible position in the join computation after which g_j can be evaluated:

$$\mathsf{gs\_index}(g_j) = \min \left\{ k \in \{1, \ldots, n\} \mid \mathsf{vars}(g_j) \subseteq \bigcup_{i=1}^{k} \mathsf{vars}(H_{\Theta(i)}) \right\}$$

If we have mode declarations for g_j, we can further refine this definition to consider only its input variables (cf. (Holzbaur et al. 2005)). We use G^k to denote the part of G that can be evaluated after having joined k heads:[4]

$$G^k = \bigwedge \{ g_j \mid 1 \leq j \leq m \wedge \mathsf{gs\_index}(g_j) = k \}$$

We distinguish between two types of host-language constraints in the guard: the first type are equality guards, the second type are other guards. The difference between both types of guards is that the first type can be evaluated in an *a priori* way: the equality can be used to do a hash table lookup which means that partner constraints that do not satisfy the equality are not even considered. Constraints of the second type can only be evaluated in an *a posteriori* way: we consider all candidate partner constraints and check whether the guard holds. We denote the type-1 guards of G^k with G^k_{eq} and the type-2 guards with $G^k_{\star}$.

[4] G^1 is the part of G that can be evaluated given only the active constraint.

6.3.2 Partial Joins

Definition 7 *Given a built-in store B, a CHR store described by $\mathsf{st}(H_i)$, a join order Θ, and an active constraint a, we define $\mathcal{J}^k_\Theta$, the partial join of k heads, as follows: if $\mathcal{D} \models B \rightarrow \bar{\exists}_\emptyset \left(G^1 \wedge H_1 = a\right)$ then $\mathcal{J}^1_\Theta = \{a\}$ and otherwise $\mathcal{J}^1_\Theta = \emptyset$. For $2 \leq k \leq n$, $\mathcal{J}^k_\Theta = \mathcal{J}^{k-1}_\Theta \bowtie H_{\Theta(k)}$, where*

$$\mathcal{J}^{k-1}_\Theta \bowtie H_{\Theta(k)} = \left\{ \bar{h} \in \mathcal{J}^{k-1}_\Theta \times \mathsf{st}(H_{\Theta(k)}) \mid \mathcal{D} \models B \rightarrow \bar{\exists}_\emptyset \left(G^k \wedge \bigwedge_{i=1}^{k} h_i = H_{\Theta(i)} \right) \right\}$$

The full join $\mathcal{J}^n_\Theta$ corresponds to the set of all tuples of matching constraints, and does not depend on the join order Θ. We use $\mathcal{E}^k_\Theta \supseteq \mathcal{J}^k_\Theta$ to denote the partial join where in the last lookup, the condition is weakened to only the type-1 part of the guard: $\mathcal{E}^1_\Theta = \{a\}$ if $\mathcal{D} \models B \rightarrow \bar{\exists}_\emptyset \left(G^1_{\mathsf{eq}} \wedge H_1 = a\right)$ and $\mathcal{E}^1_\Theta = \emptyset$ otherwise, and for $2 \leq k \leq n$:

$$\mathcal{E}^k_\Theta = \left\{ \bar{h} \in \mathcal{J}^{k-1}_\Theta \times \mathsf{st}(H_{\Theta(k)}) \mid |\mathcal{D} \models B \rightarrow \bar{\exists}_\emptyset \left(G^k_{\mathsf{eq}} \wedge \bigwedge_{i=0}^{k} h_i = H_{\Theta(i)} \right) \right\}$$

6.3.3 Cost Formula

The total cost for performing a join following join order Θ depends on the sizes of the intermediate partial joins $\mathcal{J}^k_\Theta$. The size $\left|\mathcal{J}^k_\Theta\right|$ of the partial join $\mathcal{J}^k_\Theta$ can be written as

$$\left|\mathcal{J}^k_\Theta\right| = \left|\mathcal{J}^{k-1}_\Theta\right| \cdot \sigma_{\mathsf{eq}}(k) \cdot \mu(k) \cdot \sigma_\star(k)$$

where

$$\begin{aligned} \sigma_{\mathsf{eq}}(k) &= \left|\mathcal{A}^k_\Theta\right| / \left|\mathcal{J}^{k-1}_\Theta\right| && (\textit{a priori selectivity}) \\ \mu(k) &= \left|\mathcal{E}^k_\Theta\right| / \left|\mathcal{A}^k_\Theta\right| && (\textit{multiplicity}) \\ \sigma_\star(k) &= \left|\mathcal{J}^k_\Theta\right| / \left|\mathcal{E}^k_\Theta\right| && (\textit{a posteriori selectivity}) \end{aligned}$$

and

$$\mathcal{A}^k_\Theta = \left\{ \bar{h} \in \mathcal{J}^{k-1}_\Theta \mid \exists h_k \in \mathsf{st}(H_{\Theta(k)}) : (\bar{h}, h_k) \in \mathcal{E}^k_\Theta \right\}$$

for $1 \leq k \leq n$ with $\mathcal{J}^0_\Theta = \{\epsilon\}$ with ϵ the empty tuple. Intuitively, $\sigma_{\mathsf{eq}}(k)$ represents the fraction of tuples of the partial join $\mathcal{J}^{k-1}_\Theta$ for which the *a priori* lookup of the next partner constraint is successful, that is, at least one constraint exists in $\mathsf{st}(H_{\Theta(k)})$ that satisfies the equality guard. The intuitive meaning of $\mu(k)$ is the average multiplicity of *a priori* lookups of the next partner constraint, that is, the number of constraints from $\mathsf{st}(H_{\Theta(k)})$ that correspond to a given tuple from $\mathcal{J}^{k-1}_\Theta$ with respect to G^k_{eq}, averaged over all tuples $j \in \mathcal{J}^{k-1}_\Theta$ for which at least one such constraint exists. Finally, $\sigma_\star(k)$ intuitively corresponds to the percentage of tuples from $\mathcal{J}^{k-1}_\Theta \times H_k$ that satisfy the remaining guard $G^k_\star$ given that the equality guard G^k_{eq} is satisfied.

Assuming that finding the set of partners that satisfy the *a priori* guard can be done in constant time, and assuming that evaluating the *a posteriori* guard also takes only constant time per tuple, the total cost of joining n heads using the sequence of joins $(((H_{\Theta(1)} \bowtie H_{\Theta(2)}) \bowtie H_{\Theta(3)}) \ldots \bowtie H_{\Theta(n)})$ is proportional to C^n_Θ, the sum of the *a priori* sizes of all partial joins:

$$C^n_\Theta = \sum_{j=1}^{n} \frac{|\mathcal{J}^j_\Theta|}{\sigma_\star(j)} = \sum_{j=1}^{n} \frac{\prod_{k=1}^{j} (\sigma_{\mathsf{eq}}(k) \cdot \mu(k) \cdot \sigma_\star(k))}{\sigma_\star(j)}$$

Clearly, different join orders have a different cost. It is our objective to find the join order with minimal cost.

6.4 Approximating Costs

In this section we propose static and dynamic approximations of the cost model that was defined in the previous section.

6.4.1 Static Cost Approximations

Note that the following trivial inequalities always hold:

$$0 \leq \frac{\sigma_{\mathsf{eq}}(k)}{\sigma_\star(k)} \leq 1 \leq \mu(k) \leq \left|\mathsf{st}(H_{\Theta(k)})\right|$$

In the static approach we use the upper-bound 1 as an estimate for $\sigma_{\mathsf{eq}}(k)$.

Estimating $\mu(k)$

CHR constraints have multi-set semantics in general, but in practice constraints often have set semantics or at least multiplicities that are bounded by a small constant. Moreover, often there exists a *functional dependency* between arguments of a constraint: given a CHR store S and a CHR constraint predicate $p(x_1, \ldots, x_n)$ of arity n, we say there is a functional dependency from the arguments $K = \{x_{i_1}, \ldots, x_{i_m}\}$ to the arguments $R = \{x_{j_1}, \ldots, x_{j_l}\}$ with $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ and $\{j_1, \ldots, j_l\} \subseteq \{1, \ldots, n\}$ if

$$\forall p(x_1, \ldots, x_n) \in S \wedge p(x'_1, \ldots, x'_n) \in S \wedge \\ x_{i_1} = x'_{i_1} \wedge \ldots \wedge x_{i_m} = x'_{i_m} \rightarrow x_{j_1} = x'_{j_1} \wedge \ldots \wedge x_{j_l} = x'_{j_l}$$

In words, all constraints in S that share their key arguments K, also share the values for the arguments R. We write $p(x_1, \ldots, x_n) :: K \rightsquigarrow R$. A static analysis to detect functional dependencies is given by Duck and Schrijvers (2005).

If we can derive a functional dependency for $H_{\Theta(k)}$ given the variables of the previous heads $\bigcup_{i=1}^{k-1} \mathsf{vars}(H_{\Theta(i)})$ and the equality guard G^k_{eq}, then we know that there can be at most one $H_{\Theta(k)}$ constraint for a given tuple of $\mathcal{J}^{k-1}_{\Theta}$, and so in that case, $\mu(k) = 1$. Similarly, if we can derive a generalized functional dependency with multiplicity m, that is, at most m constraints with a given key exist, then we get $1 \leq \mu(k) \leq m$ and so we can estimate $\mu(k)$ to be in the middle of that interval: $\frac{m+1}{2}$.

In general, we propose the following heuristic to estimate $\mu(k)$. Assume that $H_{\Theta(k)}$ is a constraint of arity n, and let $A = \{a_1, \ldots, a_n\}$ be its arguments. Let $FD \subseteq A$ be the subset of arguments that are declared or inferred to be ground and of a finite (and small) type. Let $f : 2^A \mapsto 2^A$ denote the reflexive transitive closure of the inferred functional dependency relation on the arguments of $H_{\Theta(k)}$.

Now we define the relation $S \overset{D}{\twoheadrightarrow} T$ with $S \subseteq A$, $T \subseteq A$ and $D \in \mathbb{N}$ as the least model of the following theory:

$$
\begin{array}{rr}
(\textbf{default}) & X \subseteq A \setminus S \rightarrow S \overset{|X|}{\twoheadrightarrow} S \cup X \\
(\textbf{transitivity}) & S \overset{D_1}{\twoheadrightarrow} T' \wedge T' \overset{D_2}{\twoheadrightarrow} T \rightarrow S \overset{D_1+D_2}{\twoheadrightarrow} T \\
(\textbf{finite domain}) & \mathit{true} \rightarrow S \overset{0}{\twoheadrightarrow} S \cup FD \\
(\textbf{func. dep.}) & \mathit{true} \rightarrow S \overset{0}{\twoheadrightarrow} f(S) \\
(\textbf{inv. func. dep.}) & X \subseteq A \setminus S \wedge |X| \geq |F| \rightarrow S \overset{|X|-|F|}{\twoheadrightarrow} S \cup X \\
 & \text{where } F = f(X) \cap S \setminus FD
\end{array}
$$

The relation models that there are $\mathcal{O}(N^D)$ constraints for any combination of the arguments in T, given the arguments in S, with N the size of the domains of arguments not in FD. We call D the degree of freedom. The **default** rule states that by default, each new argument adds one degree of freedom. The **transitivity** rule implements the transitive closure of our relation. Arguments that are functionally dependent on the already known arguments, or have a small finite domain, do not contribute to the degree of freedom (**functional dependency** and **finite domain** rules).[5] Finally, the **inverse functional dependency** rule states the following. If there is a set of arguments X that functionally determine a set of arguments F, subset of S and non-overlapping with FD, then X contributes $|X| - |F|$ to the degree of freedom given that $|X| \geq |F|$. The reasoning is as follows: for each combination of the arguments in X, there exist unique values for the arguments of F. Therefore, given that there are on average $N^{|X|}$ values for X and $N^{|F|}$ for F, we have that, again on average, $N^{|X|}/N^{|F|}$ values for X correspond to a given value for F.

[5] A better approximation would be that an argument with a finite domain of size S, contributes $1/\log_S N$ degrees of freedom.

Let $K \subseteq A$ be the subset of arguments that are given by the previous heads and the equality guard σ^k_{eq}. As a heuristic, we estimate $\mu(k)$ to be N^D, where N is an estimate of the size of the domains of arguments not in FD, and D is the minimal number for which holds that $K \overset{D}{\twoheadrightarrow} A$.

Example 6.4 Consider a constraint `c`/5 with arguments $a_1, \ldots, a_5$ and type and mode declaration `c(+int,?int,?int,+bool,?int)`.[6] Because `+bool` is the only ground finite type, we have $FD = \{a_4\}$. First assume there are no non-trivial functional dependencies for `c`/5. In an ordering like `a, c(_,_,_,_,_)`, we have $K = \emptyset$, and we get $\emptyset \overset{0}{\twoheadrightarrow} \{a_4\} \overset{4}{\twoheadrightarrow} \{a_1, a_2, a_3, a_4, a_5\}$ and so by transitivity, $D = 4$. Intuitively, there are 4 degrees of freedom to choose `c`/5: every argument with an infinite type contributes one degree of freedom. If `c`/5 has set semantics and the domain of its arguments is of size N (at runtime), then there can be only $\mathcal{O}(N^4)$ elements in `st`(`c`/5). In an ordering like `a(A), b(B,C), c(A,B,_,_,_)`, we have $K = \{a_1, a_2\}$, so now we get $\{a_1, a_2\} \overset{0}{\twoheadrightarrow} \{a_1, a_2, a_4\} \overset{2}{\twoheadrightarrow} \{a_1, a_2, a_3, a_4, a_5\}$ and $D = 2$. Intuitively, given the first two arguments, there can be only $\mathcal{O}(N^2)$ matches.

Now assume `c`/5 has a functional dependency from its first two arguments to its third argument, e.g., because of a rule

```
c(A,B,X,_,_), c(A,B,Y,_,_) ==> X = Y.
```

Consider the same ordering: `a(A), b(B,C), c(A,B,_,_,_)`. Because of the functional dependency we now have $\{a_1, a_2\} \overset{0}{\twoheadrightarrow} \{a_1, a_2, a_3\} \overset{0}{\twoheadrightarrow} \{a_1, a_2, a_3, a_4\} \overset{1}{\twoheadrightarrow} \{a_1, a_2, a_3, a_4, a_5\}$ and $D = 1$ so there are only $\mathcal{O}(N)$ matches.

Finally, consider the ordering `b(B,C), c(_,_,C,_,_)`. We have $K = \{a_3\}$ and $\{a_3\} \overset{1}{\twoheadrightarrow} \{a_1, a_2, a_3\} \overset{0}{\twoheadrightarrow} \{a_1, a_2, a_3, a_4\} \overset{1}{\twoheadrightarrow} \{a_1, a_2, a_3, a_4, a_5\}$ where the first rule applied is the **inverse functional dependency** rule. The resulting degree of freedom is 2, i.e., there are $\mathcal{O}(N^2)$ matching `c`/5 constraints given the first two arguments. The reasoning is as follows: there are $\mathcal{O}(N^2)$ combinations of the first two arguments, and every combination maps to one of the $\mathcal{O}(N)$ values of the third argument, so on average, when we fix a value of the third argument, there are only $\mathcal{O}(\frac{N^2}{N}) = \mathcal{O}(N)$ possibilities for the combination of the first two arguments. Of course this argument is only valid under some rather strong assumptions; for instance, it fails if most combinations of a_1 and a_2 result in the same value of a_3 and if precisely that value is tried. We expect to get a reasonable heuristic when the argument domains are sufficiently similar in size and the distribution of tried values for the target of the functional dependency is approximately uniform. □

[6] Here, `+`*Type* means the argument always takes a ground value of type *Type*, whereas `?`*Type* means the argument may also be non-ground.

Estimating $\sigma_\star(k)$

Let $G^k_\star = g_1 \wedge \ldots \wedge g_m$. A first assumption we make is that each of the atomic sub-guards g_i ($1 \leq i \leq m$) has a selectivity that is independent of the rest of the guard. Under this assumption, we have that $\sigma_\star(k) = \sigma_\star(g_1) \cdot \ldots \cdot \sigma_\star(g_m)$, where $\sigma_\star(g_i)$ is the selectivity of atomic guard g_i. Now we use the following values for this selectivity: $\sigma_\star(g_i) = 0.5$ if g_i is of the form $x < y$, $x > y$, $x \leq y$ or $x \geq y$; $\sigma_\star(g_i) = 0.1$ if g_i is of the form $x = y$; $\sigma_\star(g_i) = 0.9$ if g_i is of the form $x \neq y$; and $\sigma_\star(g_i) = 0.75$ otherwise. Of course this heuristic is very arbitrary and ad-hoc, and it can be extended or modified. In our heuristic, we have assumed that the values of the guard's arguments are uniformly distributed. The reasoning then is that disequalities usually hold, (arithmetic) equalities usually do not, and less-than(-or-equal) holds in half of the cases. Unknown guards are assumed to hold in 75% of the cases.

6.4.2 Dynamic Cost Approximations

In the dynamic approach, we maintain statistics about the constraint store and use them to find upper-bounds and approximations of the cost of join orders.

Worst-case Bounds

In the worst case, the trivial upper-bound of 1 is reached for both $\sigma_{\mathsf{eq}}(k)$ and $\sigma_\star(k)$. For $\mu(k)$ we can obtain a tighter upper-bound by maintaining, for every data structure which is used for constraint lookup, the maximal number of constraints per lookup key — e.g., for hash-tables this corresponds to the maximal bucket size.[7] The equality guard G^k_{eq} determines the data structure that will be used for the lookup. The maximal number of constraints per key of that data structure is hence an upper-bound for $\mu(k)$.

Approximations

Instead of maintaining the maximal number of constraints per key, we can also easily maintain the average number of constraints per key. This average is a good approximation of $\mu(k)$ if we assume that the effectively used lookup keys are representative. As an approximation of $\sigma_\star(k)$ we can either use the same heuristic as in the static case, or estimate the value by dynamically maintaining the success rate of the type-2 guards. Finally, we approximate $\sigma_{\mathsf{eq}}(k)$ as follows. We maintain the total number of distinct keys in every constraint data structure, and we compute the size of the key domain as follows: for every type used in constraint arguments, we maintain the type domain size as the number of distinct

[7] This requires a map from bucket size to bucket. We can then easily maintain the maximum as each hash table operation increases or decreases the maximum with at most one.

(ground or non-ground) values that were effectively encountered in arguments of that type. The size of a single-argument key domain is simply the domain size of its type. For multi-argument keys we use the product of the corresponding type domain sizes. In a sense, $\sigma_{\mathsf{eq}}(k)$ represents the probability that a key exists (i.e., lookup on a key is non-empty). We approximate it by dividing the number of keys by the size of the key domain. This is reasonable assuming that the keys tried are uniformly sampled from the key domain.

Hybrid Heuristic

In practice, it makes sense to use a weighted sum of the above approximation and the worst-case bound. If the approximation results in a cost C_{a} and the worst-case bound results in a cost C_{w}, we may want to optimize $(1-\alpha)\cdot C_{\mathsf{a}} + \alpha \cdot C_{\mathsf{w}}$, where α is some small constant, e.g., $\alpha = 0.05$.

6.5 Join Ordering and CHR$^{\mathrm{rp}}$

In regular CHR, we can fire the first rule instance we find. The same holds for CHR$^{\mathrm{rp}}$ given that the active occurrence only participates in rule instances of the highest priority. However, dynamic priority rules complicate the picture. In general, for rules with a dynamic priority, we may need to do some speculative work at the highest priority before we know the priority of the rule instances that extend a given partial rule match. In Section 3.6.2, we presented the compilation of dynamic priority rules by means of a source-to-source transformation. In this transformation, the speculative work consists of creating representations for partial matches in the form of r-$\mathtt{match}_j/n$ constraints. Now consider the rule

```
X :: a \ b(X,Y), c(Y) <=> d.
```

and let the a/0 constraint be active. In this case, we need to compute all the partial matches that include at least the first two heads, before we can decide which is the highest priority rule instance. Potentially, after firing the first rule instance, all these partial matches are invalidated, for example if the program also contains a rule

```
1 :: a, d <=> true.
```

Each partial match that is computed, is scheduled on the agenda which is a priority queue. In general, this operation has a higher than constant time cost. For example, when using Fibonacci heaps (Fredman and Tarjan 1987) to implement the agenda, then the scheduling cost is $\mathcal{O}(\log n)$ with n the number of scheduled items.[8] This implies that it might be better to do *more* speculative work, if this

[8] More precisely, insertion has a constant cost, but removal costs $\mathcal{O}(\log n)$ (amortized).

n	Early	Late
4096	0.73	0.01
8192	1.56	0.03
16384	3.29	0.06
32768	6.84	0.10
65536	14.03	0.21

Table 6.2: Early versus late scheduling for dynamic priority rules

results in fewer items to be scheduled. For example, assume that there are only few $\mathtt{c}/1$ constraints, then there might be more partial matches containing the first two heads of the dynamic priority rule above, than there are rule instances.

For the above example rule, we have measured the difference between scheduling early, i.e., as soon as the priority is known, versus scheduling after a full match is found. We use goals of the form

$$\{\mathtt{a}, \mathtt{b}(1,1), \ldots, \mathtt{b}(n,n), \mathtt{c}(n)\}$$

for different n. Table 6.2 shows the resulting run times in seconds for early versus late scheduling in our CHR$^{\mathrm{rp}}$ system (using the same setup as used in Section 3.8). We use Fibonacci heaps as priority queue for the agenda. The example illustrates the importance of the scheduling moment. A similar example could be constructed in which early scheduling is advantageous. We now show how the scheduling moment can be incorporated into the cost model.

For dynamic priority rules, there is a point after which partial matches are scheduled. At this point, the priority of all rule instances that extend the partial match, should be known. Let s be the number of heads after which we schedule a partial match for a given rule with a dynamic priority. We can now refine our cost formula of Section 6.3.3 as follows.

$$C'^{n}_{\Theta} = C^{n}_{\Theta} + S \cdot \prod_{k=1}^{s} (\sigma_{\star}(k) \cdot \sigma_{\mathtt{eq}}(k) \cdot \mu(k))$$

where S is the cost of scheduling an item on the agenda. In general, this cost depends on the number of items being scheduled, both by the rule at hand, and by other rules. Assuming the cost of scheduling an item with a static priority is constant[9] and the priority queue for the dynamic priorities is a Fibonacci heap, S can be approximated as the logarithm of the sum of the number of items scheduled in all dynamic priority rules according to the above formula.

Finally, it is possible that in between the moment that a partial match is scheduled, and the moment it becomes the highest priority scheduled item, one or more

[9]The distinct static priorities are known at compile time.

of its constituent constraints are removed. The cost model could be extended even further to incorporate the probability that a partial match is invalidated before it has the highest priority. We do not go into more detail here, but conclude that dynamic rule priorities and the non-trivial cost of scheduling items in a priority queue, strongly influence the optimal join order.

6.6 Finding an Optimal Join Order

This section surveys algorithms that can be used to determine an optimal join order given our cost model and approximations of its parameters. Section 6.6.1 introduces the notion of join graphs. In Section 6.6.2, an $\mathcal{O}(n \log n)$ algorithm for the special case of acyclic join graphs is presented (where n is the number of heads). Section 6.6.3 deals with the general case of cyclic join graphs.

6.6.1 Join Graphs

The join graph for a given rule is constructed as follows: the vertices correspond to the heads of the rule and the edges link heads whose variables are connected via guards.[10] Join graphs are cyclic or acyclic.

Example 6.5 Figure 6.1 shows an example of both a cyclic and an acyclic join graph. The cyclic join graph in Figure 6.1(a) corresponds to the following rule (from the timed automaton program):

```
dist(X1,Y1,D), fincl(X2,X1), fincl(Y2,Y1) \ fdist_init(X2,Y2,L) <=>
                \+ memberchk_eq(D-X1,L) | ...
```

Figure 6.1(b) shows the acyclic join graph corresponding to the following rule (from the RAM simulator program):

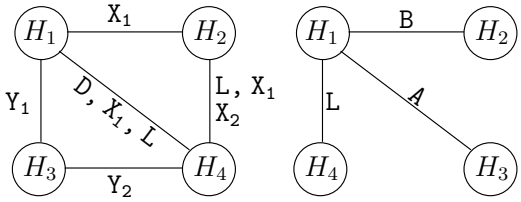
```
prog(L,move,B,A), mem(B,X) \ mem(A,_), prog_counter(L) <=> ...
```

The head indices refer to the textual order of the heads in the rules. □

6.6.2 Join Order Optimization for Acyclic Join Graphs

In this section, we show how an optimal join ordering can be found in $\mathcal{O}(n \log n)$ time for n-headed rules with an acyclic join graph using an algorithm from database query optimization (Krishnamurthy et al. 1986). An acyclic join graph can be represented as a tree with the active constraint as its root node. The algorithm presented in this section returns an optimal join order given that

[10] This includes implicit equality guards; variables are connected via a guard if the choice of a value for one of them limits the allowed values for the other ones.



(a) Cyclic (b) Acyclic

Figure 6.1: Cyclic and acyclic join graphs

1. there exists an optimal join order that respects the tree order, i.e., no head is looked up before its parent in the join tree;
2. all selections are representative: given a join and a head H extending the join, then adding or removing other heads to/from the join does not influence the selectivity and multiplicity for H.

Given a join order Θ and assume that for given j, join order Θ' is cheaper, where Θ' equals Θ except that the j^{th} and $(j+1)^{\text{th}}$ elements are switched, i.e., $\Theta' = \Theta \circ (j, j+1)$. Since under our assumptions, $\sigma_\star(i)$, $\sigma_{\mathsf{eq}}(i)$ and $\mu(i)$ only depend on the parent of the i^{th} head in the join tree, we have that

$$\begin{aligned} C_{\Theta'} - C_\Theta &= C_\Theta^{j-1} \cdot (\sigma_\star(j-1) \cdot \rho(j+1) \cdot (1 + \sigma_\star(j+1) \cdot \rho(j)) \\ &- \sigma_\star(j-1) \cdot \rho(j) \cdot (1 + \sigma_\star(j) \cdot \rho(j+1))) \leq 0 \end{aligned}$$

where $\rho(i) = \sigma_{\mathsf{eq}}(i) \cdot \mu(i)$ and $C_\Theta = C_\Theta^{|\Theta|}$. We further simplify as follows:

$$\rho(j+1) + \rho(j) \cdot \rho(j+1) \cdot \sigma_\star(j+1) \leq \rho(j) + \rho(j+1) \cdot \rho(j) \cdot \sigma_\star(j)$$

$$\rho(j+1) \cdot (1 - \rho(j) \cdot \sigma_\star(j)) \leq \rho(j) \cdot (1 - \rho(j+1) \cdot \sigma_\star(j+1))$$

Because $\rho(i) \geq 0$ for any i:

$$\frac{\rho(j+1) \cdot \sigma_\star(j+1) - 1}{\rho(j+1)} \leq \frac{\rho(j) \cdot \sigma_\star(j) - 1}{\rho(j)}$$

We expand the comparison measure towards sequences: sequences s and t can be switched if and only if ($\mathcal{J}_\Theta = \mathcal{J}_\Theta^{|\Theta|}$)

$$C_s + |\mathcal{J}_s| \cdot C_t \leq C_t + |\mathcal{J}_t| \cdot C_s$$

or, expressed in terms of *ranks* of sequences (as in (Ibaraki and Kameda 1984; Krishnamurthy et al. 1986)):

$$\mathsf{rank}(s) = \frac{|\mathcal{J}_s| - 1}{C_s} \leq \frac{|\mathcal{J}_t| - 1}{C_t} = \mathsf{rank}(t)$$

Now, consider a head H_i whose direct parent H has a higher rank. We can show that only descendants of this parent are allowed between H and H_i. Assume that another head H_j appears in between H and H_i in an optimal order. Since H_j is not a descendant of H, it can be placed before H. Assume it has a lower rank than H, then we can move it from in between H and H_i to before H and the resulting cost is smaller. If it has a higher rank than H, then it also has a higher rank than H_i and it can be moved to after H_i, resulting again in a smaller cost. In conclusion, if a head has lower rank than its parent, then only descendants of this parent, can be in between the head and its parent in an optimal join order.

Below is the algorithm as given by Krishnamurthy et al. (1986), which outputs a single chain[11] representing an optimal join order. Let there be given a join tree T.

1. If T is a single chain then stop.
2. Find a subtree of T, all of whose children are chains. Let r be the root of this subtree.
3. Merge the chains based on ranks such that the resulting single chain is non-decreasing on the rank.
4. Normalize the root r of the subtree as follows: while the rank of r is greater than that of its immediate child c: replace r and c by a new node representing the subchain r followed by c.
5. Go to 1.

Example 6.6 Consider the following rule where the first head is active:

```
h1(X1,Y1), h2(X1,X2), h3(Y1,Y2), h4(Y2), h5(X2) ==> ...
```

It leads to the (directed) join graph shown in Figure 6.2, in which an edge to H_i is annotated with $\sigma_{\mathsf{eq}}(H_i) \cdot \mu(H_i)$.

Assume that we start with the textual order as initial join order:

$$C_{[1,2,3,4,5]} = 4 + 4 \cdot 2 + 4 \cdot 2 \cdot 3 + 4 \cdot 2 \cdot 3 \cdot 1 = 60$$

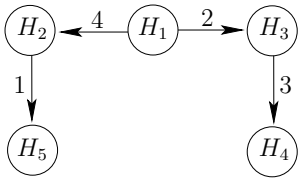
Since $\mathsf{rank}(H_5) < \mathsf{rank}(H_2)$, but H_2 should be before H_5 according to the tree order, we have that no heads should be scheduled in between H_2 and H_5. Indeed, since $\mathsf{rank}(H_3) < \mathsf{rank}(H_2)$ it is cheaper to put H_3 before H_2:

$$C_{[1,3,2,4,5]} = 2 + 2 \cdot 4 + 2 \cdot 4 \cdot 3 + 2 \cdot 4 \cdot 3 \cdot 1 = 58$$

and similarly for H_4 and H_2:

$$C_{[1,3,4,2,5]} = 2 + 2 \cdot 3 + 2 \cdot 3 \cdot 4 + 2 \cdot 3 \cdot 4 \cdot 1 = 56$$

[11] A chain is a tree in which each node has at most one child.



Head	$\lvert\mathcal{J}\rvert$	C	Rank
H_2	4	4	3/4
H_3	2	2	1/2
H_4	3	3	2/3
H_5	1	1	0

Figure 6.2: Acyclic join graph and ranks

Now H_2 is directly before H_5 and we combine both into a new node with a combined rank of $\mathsf{rank}([H_2, H_5]) = 3/8$. Now we have that $\mathsf{rank}([H_2, H_5]) < \mathsf{rank}(H_4)$ and so we can switch them:

$$C_{[1,3,2,5,4]} = 2 + 2 \cdot 4 + 2 \cdot 4 \cdot 1 + 2 \cdot 4 \cdot 1 \cdot 3 = 42$$

and similarly for $[H_2, H_5]$ and H_3:

$$C_{[1,2,5,3,4]} = 4 + 4 \cdot 1 + 4 \cdot 1 \cdot 2 + 4 \cdot 1 \cdot 2 \cdot 3 = 40$$

Now all heads (sequences) are sorted by rank and the join order is optimal. □

6.6.3 Join Order Optimization for Cyclic Join Graphs

While we have an efficient algorithm for finding an optimal join order for acyclic join graphs, the same optimization problem for general graphs has been proven to be NP-complete (Ibaraki and Kameda 1984). In this subsection, we review some exact and approximative solutions to the join ordering problem for cyclic graphs.

Exact Algorithm

An exact algorithm considers all permutations of the heads, but can make use of the fact that given an optimal join order $\Theta = s +\!\!+ t$, s is an optimal join order for the subset of heads appearing in it:

$$C_{s+\!\!+t} = C_s + |\mathcal{J}_s| \cdot \sum_{j=|s|}^{n-1} \prod_{k=|s|}^{j} (\sigma_\star(k) \cdot \sigma_{\mathsf{eq}}(k+1) \cdot \mu(k+1))$$

The second part of the sum is independent of the order of the heads in s.

The task at hand is to find the shortest path in a weighted directed acyclic graph (DAG) of the form shown in Figure 6.3 for a 4-headed rule. In general, for an n-headed rule, the graph has 2^n nodes and $n \cdot 2^{n-1}$ edges. We can now use a single source shortest path algorithm for DAGs (based on topological order) which

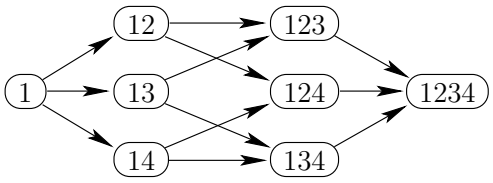


Figure 6.3: DAG for finding an optimal join order using a shortest path algorithm

has a runtime proportional to the number of edges in the graph, i.e., $\mathcal{O}(n \cdot 2^n)$. An often used heuristic is to avoid Cartesian products by postponing choosing heads which have no variables in common with the partial join computed so far, i.e., only those nodes connected to the partial join in the join graph are considered unless this is not possible (see e.g. (Selinger et al. 1979)). This can however lead to suboptimal solutions.

Example 6.7 As an example of a suboptimal solution by postponing Cartesian products, consider the rule:

```
a(X), b(X,Y), c(Y) ==> ...
```

Assume a constraint store containing many `b`/2 constraints for any given value of its first argument, and few `c`/1 constraints. Let the first head be the active constraint. The second head has a variable in common with the active constraint whereas the third head has not. However, it is more interesting to lookup the third head first, as this drastically limits the amount of `b`/2 constraints considered for the second head. □

As an improvement on the simple shortest path algorithm, we can use the A* algorithm (Hart et al. 1968) by making an estimate of the cost of joining the remaining heads, given a partial join. In the K.U.Leuven CHR system, a similar approach was taken, although the distinction between the exact cost of what has been joined so far, and the heuristic estimate of what remains to be joined, has been lost over time and currently a more simple form of best-first search is used. Now, clearly, we want the heuristic to be computationally cheap. A very simple estimate for the remaining cost is the size of the total join, but this size is the same for all join orders, so it does not contribute to the search. If we make the assumption of representative selection, as we did for the case of acyclic join graphs, as well as in our estimates of actual selectivity and multiplicity, then we can use the following formula:

$$C_{s+\!+t} \geq C_s + |\mathcal{J}_s| \cdot \sum_{j=|s|}^{n-1} \prod_{k=|s|}^{j} \left(\sigma'_\star(k) \cdot \sigma'_{\mathsf{eq}}(k+1) \cdot \mu'(k+1)\right)$$

where $\sigma'_\star$, σ'_{eq} and μ' are defined similarly as $\sigma_\star$, σ_{eq} and μ, but also take into account the information from the remaining heads. The join order that minimizes this underestimation (and hence is an underestimation for any order of the remaining heads t) is the one in which the heads of t are ordered by increasing $\sigma'_{\mathsf{eq}} \cdot \mu' \cdot \sigma'_\star$.[12] Moreover, we have that an optimal order for t projected on a subset of t, is also an optimal order for this subset. This implies that we can compute an optimal order for the full join once and only need to select the relevant terms during search. The complexity of calculating the heuristic lower bound during search is hence proportional to the number of heads n (not including the initial sorting phase before the search).

Since the exact algorithm has an exponential time and space complexity, and the A* algorithm is not better in the worst case, we look at some approximative alternatives in the remainder of this subsection.

Minimal Spanning Tree

In (Krishnamurthy et al. 1986), cyclic join graphs are first made acyclic by constructing a Minimal Spanning Tree (MST) and applying the algorithm for acyclic graphs on the resulting join tree. The edge weights used during MST construction are the selectivities. In (Swami and Iyer 1993), a more complex solution is proposed. It consists of applying the algorithm for acyclic join graphs to a (random) spanning tree, followed by a phase of iterative improvement in which pairs of heads are switched. This process is repeated multiple times with randomized starting points. In any case, an optimal solution found using any spanning tree is not necessarily optimal for the complete join graph.

Randomized Algorithms

Given that we use approximations for $\sigma_\star$, σ_{eq} and μ, an optimal join order found using exact algorithms is in fact still an approximation. This motivates the use of randomized algorithms that give approximatively optimal solutions to the inexact optimization problem. An overview of such randomized algorithms can be found in (Steinbrunn et al. 1997). They include iterative improvement and simulated annealing starting from random initial join orders, as well as the application of genetic algorithms.

6.6.4 Optimization Cost versus Join Cost

While some solutions for the optimization problem, in particular the exact algorithm for cyclic join graphs, may appear to be prohibitively expensive, the opti-

[12] Implicitly, we consider the join graph for the heads in t to be such that all heads are linked to a common root node, and no two heads are directly linked. We can use the algorithm of Section 6.6.2 in this case.

mization cost should always be compared to the cost of actually performing the join. Assume for example that in an optimal join order, $\sigma_\star(k-1)\cdot\sigma_{\mathsf{eq}}(k)\cdot\mu(k) \geq 2$ for all $k \in \{1,\ldots,n\}$, then the cost of performing the join is $\Omega(2^n)$ and so the optimization cost may not be such a problem. Moreover, one should compare the optimization cost with the join cost for the best plan found so far (i.e., after a previous optimization step), rather than with that of an optimal join order. Therefore, one approach is to first optimize using a fast but suboptimal algorithm (e.g., a greedy algorithm), and then, only if the expected optimization cost of using an exact algorithm is less than the join cost of the plan resulting from the first optimization step, we optimize further using such an algorithm.

6.7 Discussion

6.7.1 First Few Answers

As already pointed out in Section 6.3, our cost model is based on the assumption that all applicable rule instances eventually fire. In many cases, this is obviously not true. Consider for example a simplification or simpagation rule in which the active constraint is removed. Clearly, only the first rule instance found will fire. Therefore, it can be advantageous to optimize for finding the first answer. In this section, we give an alternative cost model, which represents the cost for finding the first full match. This model is based on (Bayardo and Miranker 1996).

Let $p(i)$ denote the probability that a tuple from the partial join up to the i^{th} head (according to some join order Θ) can be extended to a full join tuple. We recursively define $p(i)$ as follows:

$$p(i) = \sigma_{\mathsf{eq}}(i) \cdot (1 - (1 - p(i+1))^{\mu(i)\cdot\sigma_\star(i)})$$

It is the probability that the tuple can be extended by at least one head ($\sigma_{\mathsf{eq}}(i)$), times the probability that at least one of the tuples of $i+1$ heads survives (i.e., one minus the probability that all fail). The total cost of finding the first answer can then be written as follows:

$$C = \sum_{i=1}^{n} \frac{1}{p(i)}$$

Here we assume that $1/p(i)$ never exceeds $|\mathcal{J}_i|$. The reasoning is as follows: $p(i)\cdot|\mathcal{J}_i|$ tuples out of $|\mathcal{J}_i|$ will participate in a full match. Under the assumption that these tuples are uniformly distributed amongst all tuples, we will need to skip $1/p(i)$ tuples before finding the first tuple that takes part in a full match.

For more general types of rules, we can make a weighted sum of the cost model introduced here, and the one from Section 6.3.

6.7.2 Cyclic and Acyclic Join Graphs

Table 6.3 shows for some example programs the number of n-headed rules ($n \in \{1, \ldots, 6\}$) and in between brackets the number of rules with cyclic join graphs.

Name	1	2	3	4	5	6	Total
EU Car Rental	5	4	2 (0)	5 (4)	2 (2)		18 (6)
Hopcroft	10	9	4 (1)	1 (1)			24 (2)
Monkey & Bananas	1	7	15 (7)	2 (0)			25 (7)
RAM Simulator	1	2	3 (0)	5 (0)	2 (0)		13 (0)
Timed Automaton		11	10 (9)	3 (3)			24 (12)
Type Inference	26	48	13 (7)	6 (5)	4 (4)		97 (16)
Well-founded Semantics	3	25	8 (1)	4 (2)	1 (1)	2 (2)	43 (6)
Total	46	106	55 (25)	26 (15)	9 (7)	2 (2)	244 (49)

Table 6.3: Cyclic and acyclic join graphs in example programs

Not every edge in a join graph has the same importance as illustrated by the next example.

Example 6.8 Consider the rule (from the type inference program)

```
st(S,T1), st(S,T2) \ rt(T1,R1), rt(T2,R2) <=> R1 \== R2 | ...
```

The disequality guard creates a cycle in the join graph. However, its influence on the optimal join order is expected to be minimal, i.e., we can assume that it is better to lookup $\texttt{rt}(\texttt{T}_2, \texttt{R}_2)$ via $\texttt{T}_2$ than using $\texttt{R}_1$ `\==` $\texttt{R}_2$ given $\texttt{R}_1$. □

Example 6.9 Consider the rule (from the well-founded semantics program)

```
true(At) \ neg(At,Cl), aclause(Cl,OAt),
                nbulit(Cl,_), nbplit(Cl,_), nbucl(OAt,N) <=> ...
```

The join graph is cyclic because variable `Cl` appears in more than two heads. However, once we have `Cl`, looking up another head containing it does not give us any new information. Under the assumption of representative selection that we made for the approximations of the cost model parameters, we can turn the join graph into a rooted join tree (one for every possible active constraint). Figure 6.4 depicts the original join graph (heads are numbered in textual order) and its transformation into a rooted join tree.

The idea is as follows: if the join graph contains a *clique* such that every two nodes in the clique share exactly the same variables, and moreover it holds that from any node outside of the clique, there exists at most one path to any node of the clique, then given a root node, there exists a unique entry point into the clique (possibly the root node belongs to the clique itself) and we can remove all edges in the clique except those connected to this entry point. □

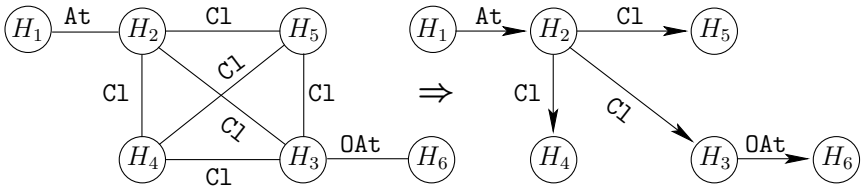


Figure 6.4: Transformation of a join graph with clique into a rooted join tree

However, if some nodes of a clique share more variables than others, we cannot make the join graph acyclic.

Example 6.10 Consider the rule (from the monkey & bananas program)

```
goal(a,h,A,B,C), phys_object(A,E,l,n,F,G,H), monkey(E,J,A) ==> ...
```

Variable `A` appears in all three heads, but the second and third head also share variable `E`. Hence we cannot reduce the join graph to a rooted join tree. □

In the example programs of Table 6.3, only 10 of the 49 cyclic graphs remain cyclic after excluding the above two types of cyclicity.

6.7.3 Join Ordering in Current CHR Implementations

Current CHR implementations generally follow (a variation of) the HAL heuristic (Holzbaur et al. 2005; Duck 2005), if they apply join ordering at all (e.g., Haskell CHR or the ECL^{i}PS^e `ech` library (Wallace et al. 1997) use the textual order of heads). The K.U.Leuven CHR system uses a heuristic that was originally based on the HAL heuristic, but has become more complex over time due to incremental compiler changes.

The cost model for join ordering in the HAL CHR system is based on the assumption that each constraint argument ranges over a domain of the same size s. Let there be given a partial join $\mathcal{J}^i_\Theta$ and let w_i be the number of arguments of $H_{\Theta(i+1)}$ that are not fixed by $\mathcal{J}^i_\Theta$ (degrees of freedom). The worst case size of $\mathcal{J}^{i+1}_\Theta$ is $\left|\mathcal{J}^i_\Theta\right| \cdot s^{w_i} = s^{w_1+\ldots+w_i}$ In order to eliminate the domain size s, the cost of the total join is approximated by $(n-1)\cdot w_1 + (n-2)\cdot w_2 + \ldots + 1 \cdot w_{n-1}$ which is in fact the logarithm (to base s) of $\left|\mathcal{J}^1_\Theta\right| \cdot \left|\mathcal{J}^2_\Theta\right| \cdot \ldots \cdot |\mathcal{J}^n_\Theta|$ whereas the real cost would be the sum of these partial join sizes. The weights relate to our approach as follows: $\mu(i) \approx s^{w_i}$.

The number of degrees of freedom is reduced by functional dependencies and explicit equality guards. For other guards, the reduction depends on the determinism of the guard predicate. For example, if the predicate is declared as `semidet`, then the degree of freedom is reduced by 0.25 per fixed argument. The motivation

is a guard `X>Y` which would remove 0.5 degrees of freedom. Note though that this implies a cost $s^{1.5}$ for a constraint `c(X,Y)` with guard `X>Y` whereas a more accurate estimate would be $s^2/2$.

Finally, we note that although it is mentioned in (Duck 2005; Holzbaur et al. 2005) that we often only need the first full match, the HAL cost model does not reflect this concern.

6.8 Conclusion

We summarize the contributions of this chapter. We have given a new cost model for matching multi-headed rules that is more realistic and flexible compared to earlier work. Our work is the first to consider runtime statistics such as cardinalities and selectivities in the context of CHR. We have shown how these statistics can be used as estimates for the parameters of our cost model. We have looked at various algorithms for join order optimization from the database literature, and ported them to the CHR context. In particular, we looked at an efficient algorithm for rules with acyclic join graphs. Furthermore, we have shown that many cyclic join graphs can in fact be made acyclic under certain assumptions. Finally, we have discussed the issue of optimization for the first few answers, and have given an alternative cost model for this case.

Future work

In this chapter, we have only looked at the theoretical side of the join ordering problem. Although it is already clear that a good join order, and even dynamic join ordering can give rise to a better runtime complexity, it remains an open question whether it is worth the extra effort in typical problems. An implementation of the proposed join ordering schemes may give some answers concerning the practical side of the join ordering problem. Such an implementation will also allow for experimentation with ideas such as computing optimal join orders in parallel using a different thread, and hybrid static and dynamic join ordering (where the dynamic search space is reduced at compile time, e.g., based on the join graph or functional dependency analysis). Other issues that need further investigation include the trade-off between optimization cost and join cost, using profiling information for static join ordering, and other criteria (apart from those given in Section 6.7.2) to reduce a cyclic join graph to an acyclic one.

Chapter 7

General Conclusion

In the introduction, we stated the goal of this thesis as the design, analysis, and implementation of language concepts and constructs to support flexible, high-level, and efficient execution control in CHR. In this final chapter, we summarize our contributions towards this goal. We also give directions for future research.

7.1 Contributions

We summarize our contributions below.

- In Chapter 3, we introduced CHR$^{\text{rp}}$: CHR extended with user-defined rule priorities. This language extension allows for a much more high-level and flexible form of execution control than previously available, in particular w.r.t. the refined operational semantics of regular CHR. We have shown by example how CHR$^{\text{rp}}$ can be used to express advanced propagation strategies in a concise way. We established correspondence results between, on the one hand, CHR$^{\text{rp}}$ programs and derivations under its operational semantics (ω_p), and on the other hand, corresponding CHR programs (without the priorities) and derivations under the theoretical operational semantics (ω_t) of CHR. We discussed how confluence results for CHR map onto CHR$^{\text{rp}}$ and linked confluence under the ω_p semantics to the concept of observable confluence. On the practical side, it is shown how typical programming patterns based on the refined operational semantics of CHR, transfer to CHR$^{\text{rp}}$. A basic compilation schema for CHR$^{\text{rp}}$ is presented and optimizations on this schema are proposed. The resulting CHR$^{\text{rp}}$ implementation is evaluated empirically and compared with the state-of-the-art K.U.Leuven CHR system. In particular, it is shown that the proposed optimizations are effective, and that our implementation is already competitive w.r.t. K.U.Leuven CHR while offering a much more high-level form of execution control.

- In Chapter 4, we have proposed a new framework for execution control in CHR with disjunction ($\text{CHR}^\vee$), which combines $\text{CHR}^\vee$ and CHR^{rp}, and extends it with prioritized search tree alternatives (branch priorities). The result is a uniform solution to execution control for CHR that allows both controlling the propagation strategy (by means of rule priorities) and the search strategy (by means of branch priorities). The correspondence between derivations in the framework (called $\text{CHR}_\vee^{\text{brp}}$) and derivations under the theoretical operational semantics of $\text{CHR}^\vee$ is shown. Next, it is demonstrated how various common search strategies can easily be expressed in $\text{CHR}_\vee^{\text{brp}}$. This includes both uninformed strategies such as depth-first or breadth-first, as well as informed strategies such as best-first search. Moreover, we give two examples of how different strategies can be combined. Finally, we present an extended operational semantics for $\text{CHR}_\vee^{\text{brp}}$ to support conflict-directed backjumping (CBJ). We give a condition on programs that ensures that no answers are missed because of the pruning done by the CBJ algorithm, and show how CBJ solves the problem of potential non-termination in our $\text{CHR}_\vee^{\text{brp}}$ implementation of iterative deepening search.

- In Chapter 5, we investigated the correspondence between Ganzinger and McAllester's Logical Algorithms formalism (LA) and CHR^{rp}. The former consists of the Logical Algorithms language, which is a theoretical bottom-up logic programming language with prioritized rules, and a meta-complexity result that allows for the derivation of a LA program's time complexity in terms of the number of (partial) rule firings and assertions. We have presented translation schemas from the LA language to CHR^{rp}, and from a subset of CHR^{rp} to LA. It is shown that the translated programs are operationally equivalent to the originals. The translation from LA to CHR^{rp} allows us to execute the LA programs using our CHR^{rp} implementation, whereas the translation from CHR^{rp} to LA allows us to apply the LA meta-complexity theorem on (a subset of) CHR^{rp}. As a final step, we propose an alternative implementation of CHR^{rp} based on the LA compilation schema, extended towards non-ground constraints. This implementation gives rise to a new and strong meta-complexity result for CHR^{rp} that subsumes the LA result and, save some exceptions, gives better bounds than previous meta-complexity results for (regular) CHR.

- In Chapter 6, we analyzed the join ordering problem in $\text{CHR}^{(\text{rp})}$. This problem consists of finding an optimal order in which to look up constraints (joining) while matching a multi-headed rule. We have proposed a new model for the cost of joining, that is more realistic than previous models. We consider both the case that we need all instances of a given rule, and the case that we need only the first rule instance. Each case leads to a different cost model, and both models can be combined to get an average case estimate.

An important novelty of our approach is that we also consider join order optimization at runtime, during which we can make use of more accurate estimates of the cost model's parameters. In the context of CHR^{rp}, dynamic priority rules may require some joining work before the actual priority of rule instances extending the partial join, is known. As soon as this priority is known, the partial rule instance can be scheduled on the agenda. However, the cost of this scheduling operation makes that it is sometimes better to do more speculative work before scheduling. This observation leads to a further refinement of our cost model for CHR^{rp}. Finally, we have ported some important join order optimization algorithms from the database world to the CHR context. In particular, in the case of rules with an acyclic join graph, we can use an algorithm to find an optimal join order in time linearithmic ($n \log n$) in the number of heads. However, in general, the join-ordering problem is NP-complete.

7.2 Future Work

In this final section, we give some directions for future research. See also the future work sections of Chapters 3, 4, 5 and 6.

7.2.1 Future Implementation Work

During the course of this PhD, we developed an implementation of CHR^{rp} for Prolog. Some other proposals made in this thesis have not been implemented yet. A first of these is the $\text{CHR}^{\text{brp}}_{\vee}$ framework. Prolog CHR implementations are also $\text{CHR}^{\vee}$ implementations by means of Prolog's built-in search which is based on a trailing mechanism. However, in these implementations, the search strategy is fixed to left-to-right depth-first search. A $\text{CHR}^{\text{brp}}_{\vee}$ implementation needs to be able to jump between nodes in different parts of the search tree. In this generalized setting, a simple trailing mechanism is not sufficient, and other strategies to restore states (e.g., copying and recomputation) need to be considered as well. Moreover, to decide which of the alternatives has the highest priority, some form of priority queue is to be used. The best choice for such a queue depends on the search strategy being implemented.

In Chapter 5, we proposed an alternative implementation for CHR^{rp} that offers strong complexity guarantees. This implementation is based on the scheduling algorithm presented in (De Koninck 2007). We currently have an implementation of the scheduler in Java. The remaining part of the alternative CHR^{rp} compilation schema should be relatively straightforward to implement. The resulting system then uses a lazy form of RETE matching, and it would be interesting to compare this alternative style of matching with the current implementations of both CHR and CHR^{rp}. Moreover, the complexity guarantees of our scheduling data structure

may be stronger than needed in practice, and a simplified version of it may be more competitive in the average case.

7.2.2 Other Open Problems

In Section 3.4.1, we discussed confluence under CHR$^{\text{rp}}$'s ω_p semantics. While this discussion gave some interesting insights on the problem, a practical confluence test is still lacking. In Chapter 5, we proposed a systematic approach to derive the time complexity of CHR$^{\text{rp}}$ programs. Other important program properties which we have not studied in detail include space complexity and termination.

In this thesis, we have dealt with the problem of execution control in CHR. Apart from this control aspect, also the way in which the logic of a constraint solver is expressed in CHR could be improved. One result toward this goal is given by Van Weert et al. (2008) where CHR is extended with aggregates. Also noteworthy is the ACD Term Rewriting language (Duck et al. 2006) (ACDTR) which subsumes both CHR and AC term rewriting. One interesting feature of ACDTR is that all constraints are not necessarily part of one single (flat) conjunction as in CHR.

Appendix A

Example of Compiled Code

In this appendix, we illustrate the CHR^{rp} compilation process on the `leq` program, which is given again below for convenience.

```
1 :: reflexivity  @ leq(X,X) <=> true.
1 :: antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
1 :: idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
2 :: transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

The code below results from the basic compilation schema as outlined in Section 3.6.

```
leq(X,Y) :-
        next_free_id(Id),
        empty_history(Hist),
        S = suspension(Id,state(alive),Hist,X,Y,_,_,_,_),
        schedule(1,'leq/2_prio_1_occ_1_1'(S)),
        schedule(2,'leq/2_prio_2_occ_1_1'(S)),
        attach_to_vars(leq/2,S),
        insert_into_index(leq/2,[],S),
        insert_into_index(leq/2,[1],S),
        insert_into_index(leq/2,[2],S),
        insert_into_index(leq/2,[1,2],S).

'leq/2_prio_1_occ_1_1'(S1) :-
        (   S1 = suspension(_,State,_,X,Y,_,_,_,_),
            arg(1,State,alive),
            X == Y
        ->  setarg(1,State,dead),
            remove_from_index(leq/2,[],S1),
            remove_from_index(leq/2,[1],S1),
```

```
            remove_from_index(leq/2,[2],S1),
            remove_from_index(leq/2,[1,2],S1),
            detach_from_vars(leq/2,S1),
            true, % body
            check_activation(1)
        ;   'leq/2_prio_1_occ_2_1'(S1)
        ).

% ... leq/2 priority 1 occurrences 2 - 4

'leq/2_prio_1_occ_5_1'(S1) :-
        (   S1 = suspension(_,State,_,X,Y,_,_,_,_),
            arg(1,State,alive),
            index_lookup(leq/2,[1,2],[X,Y],S2s)
        ->  'leq/2_prio_1_occ_5_2'(S2s,S1)
        ;   true
        ).
'leq/2_prio_1_occ_5_2'([S2|S2s],S1) :-
        (   S2 = suspension(_,State2,_,X2,Y2,_,_,_,_),
            arg(1,State2,alive), S2 \= S1,
            S1 = suspension(_,State1,_,X1,Y1,_,_,_,_),
            X1 == X2, Y1 == Y2
        ->  setarg(1,State2,dead),
            remove_from_index(leq/2,[],S2),
            remove_from_index(leq/2,[1],S2),
            remove_from_index(leq/2,[2],S2),
            remove_from_index(leq/2,[1,2],S2),
            detach_from_vars(leq/2,S2),
            true, % body
            check_activation(1),
            (   arg(1,State1,alive)
            ->  'leq/2_prio_1_occ_5_2'(S2s,S1)
            ;   true
            )
        ;   'leq/2_prio_1_occ_5_2'(S2s,S1)
        ).
'leq/2_prio_1_occ_5_2'([],_).

'leq/2_prio_2_occ_1_1'(S1) :-
        (   S1 = suspension(_,State,_,Y,Z,_,_,_,_),
            arg(1,State,alive),
            index_lookup(leq/2,[2],[Y],S2s)
        ->  'leq/2_prio_2_occ_1_2(S2s,S1)
        ;   'leq/2_prio_2_occ_2_1(S1)
        ).
'leq/2_prio_2_occ_1_2'([S2|S2s],S1) :-
```

```
        (   S2 = suspension(Id2,State2,Hist2,X,Y2,_,_,_,_),
            arg(1,State2,alive), S2 \= S1,
            S1 = suspension(Id1,State1,Hist1,Y1,Z,_,_,_,_),
            Y1 == Y2,
            HistoryTuple = t(transitivity,[Id2,Id1]),
            not_in_history(Hist1,HistoryTuple),
            not_in_history(Hist2,HistoryTuple)
        ->  add_to_history(Hist1,HistoryTuple)
            leq(X,Z), % body
            check_activation(2),
            (   arg(1,State1,alive)
            ->  'leq/2_prio_2_occ_1_2'(S2s,S1)
            ;   true
            )
        ;   'leq/2_prio_2_occ_1_2'(S2s,S1)
        ).
'leq/2_prio_2_occ_1_2'([],S1) :- 'leq/2_prio_2_occ_2_1'(S1).

% ... leq/2 priority 2 occurrence 2
```

Finally, the code below results from applying the reduced activation checking, inline activation, and late indexing optimizations, as presented in Section 3.7.

```
leq(X,Y) :-
        next_free_id(Id),
        empty_history(Hist),
        S = suspension(Id,state(alive),Hist,X,Y,_,_,_,_),
        schedule(1,'leq/2_prio_1_occ_1_1'(S)),
        attach_to_vars(leq/2,S).

'leq/2_prio_1_occ_1_1'(S1) :-
        (   S1 = suspension(_,State,_,X,Y,_,_,_,_),
            arg(1,State,alive),
            X == Y
        ->  setarg(1,State,dead),
            remove_from_index(leq/2,[],S1),
            remove_from_index(leq/2,[1],S1),
            remove_from_index(leq/2,[2],S1),
            remove_from_index(leq/2,[1,2],S1),
            detach_from_vars(leq/2,S1),
            true, % body
            true % reduced activation checking
        ;   'leq/2_prio_1_occ_2_1'(S1)
        ).

% ... leq/2 priority 1 occurrences 2 - 4
```

```
'leq/2_prio_1_occ_5_1'(S1) :-
        (   S1 = suspension(_,State,_,X,Y,_,_,_,_),
            arg(1,State,alive),
            index_lookup(leq/2,[1,2],[X,Y],S2s)
        ->  'leq/2_prio_1_occ_5_2'(S2s,S1)
        ;   insert_into_index(leq/2,[1,2],S). % late indexing
            schedule(2,'leq/2_prio_1_occ_1_1'(S))
        ).
'leq/2_prio_1_occ_5_2'([S2|S2s],S1) :-
        (   S2 = suspension(_,State2,_,X2,Y2,_,_,_,_),
            arg(1,State2,alive), S2 \= S1,
            S1 = suspension(_,State1,_,X1,Y1,_,_,_,_),
            X1 == X2, Y1 == Y2
        ->  setarg(1,State2,dead),
            remove_from_index(leq/2,[],S2),
            remove_from_index(leq/2,[1],S2),
            remove_from_index(leq/2,[2],S2),
            remove_from_index(leq/2,[1,2],S2),
            detach_from_vars(leq/2,S2),
            true, % body
            true % reduced activation checking
            (   arg(1,State1,alive)
            ->  'leq/2_prio_1_occ_5_2'(S2s,S1)
            ;   true
            )
        ;   'leq/2_prio_1_occ_5_2'(S2s,S1)
        ).
'leq/2_prio_1_occ_5_2'([],S) :-
        insert_into_index(leq/2,[1,2],S). % late indexing
        schedule(2,'leq/2_prio_1_occ_1_1'(S))

'leq/2_prio_2_occ_1_1'(S1) :-
        (   S1 = suspension(_,State,_,Y,Z,_,_,_,_),
            arg(1,State,alive),
            index_lookup(leq/2,[2],[Y],S2s)
        ->  'leq/2_prio_2_occ_1_2(S2s,S1)
        ;   'leq/2_prio_2_occ_2_1(S1)
        ).
'leq/2_prio_2_occ_1_2'([S2|S2s],S1) :-
        (   S2 = suspension(Id2,State2,Hist2,X,Y2,_,_,_,_),
            arg(1,State2,alive), S2 \= S1,
            S1 = suspension(Id1,State1,Hist1,Y1,Z,_,_,_,_),
            Y1 == Y2,
            HistoryTuple = t(transitivity,[Id2,Id1]),
            not_in_history(Hist1,HistoryTuple),
```

```
            not_in_history(Hist2,HistoryTuple)
        ->  add_to_history(Hist1,HistoryTuple)
            % inline activation of leq(X,Z)
            next_free_id(Id),
            empty_history(Hist),
            S = suspension(Id,state(alive),Hist,X,Y,_,_,_,_),
            attach_to_vars(leq/2,S),
            'leq/2_prio_1_occ_1_1'(S),
            true, % reduced activation checking
            (   arg(1,State1,alive)
            ->  'leq/2_prio_2_occ_1_2'(S2s,S1)
            ;   true
            )
        ;   'leq/2_prio_2_occ_1_2'(S2s,S1)
        ).
'leq/2_prio_2_occ_1_2'([],S1) :- 'leq/2_prio_2_occ_2_1'(S1).

% ... leq/2 priority 2 occurrence 2
```

Bibliography

Abdennadher, S. 1997. Operational semantics and confluence of constraint propagation rules. In *3rd International Conference on Principles and Practice of Constraint Programming*, G. Smolka, Ed. Lecture Notes in Computer Science, vol. 1330. Springer, 252–266.

Abdennadher, S. 2000. A language for experimenting with declarative paradigms. In *2nd Workshop on Rule-Based Constraint Reasoning and Programming*.

Abdennadher, S. 2001. Rule based constraint programming: Theory and practice. Habilitation, Institut für Informatik, Ludwig-Maximilians-Universität München.

Abdennadher, S. and Christiansen, H. 2000. An experimental CLP platform for integrity constraints and abduction. In *4th International Conference on Flexible Query Answering Systems*, H. L. Larsen, J. Kacprzyk, S. Zadrozny, T. Andreasen, and H. Christiansen, Eds. Springer, 141–152.

Abdennadher, S., Krämer, E., Saft, M., and Schmauss, M. 2002. JACK: A Java constraint kit. In *International Workshop on Functional and (Constraint) Logic Programming*, M. Hanus, Ed. Electronic Notes in Theoretical Computer Science, vol. 64. Elsevier.

Abdennadher, S. and Marte, M. 2000. University course timetabling using Constraint Handling Rules. *Applied Artificial Intelligence: Special Issue on Constraint Handling Rules 14,* 4, 311–325.

Abdennadher, S., Saft, M., and Will, S. 2000. Classroom assignment using constraint logic programming. In *2nd International Conference and Exhibition on Practical Application of Constraint Technologies and Logic Programming*.

Abdennadher, S. and Schütz, H. 1998. CHR$^{\vee}$: A flexible query language. In *3rd International Conference on Flexible Query Answering Systems*, T. Andreasen, H. Christiansen, and H. L. Larsen, Eds. Lecture Notes in Computer Science, vol. 1495. Springer, 1–14.

Aguilar-Solis, D. and Dahl, V. 2004. Coordination revisited – a Constraint Handling Rule approach. In *9th Ibero-American Conference on AI*, C. Lemaître, C. A. Reyes, and J. A. Gozáles, Eds. Lecture Notes in Computer Science, vol. 3315. Springer, 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 *19th Workshop on (Constraint) Logic Programming*, A. Wolf, T. Frühwirth, and M. Meister, Eds. Ulmer Informatik-Berichte, vol. 2005-01. Universität Ulm, 111–122.

Alberti, M., Gavanelli, M., Lamma, E., Mello, P., and Milano, M. 2005. A CHR-based implementation of known arc-consistency. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules 5,* 4 & 5, 419–440.

Alves, S. and Florido, M. 2002. Type inference using Constraint Handling Rules. In *International Workshop on Functional and (Constraint) Logic Programming*, M. Hanus, Ed. Electronic Notes in Theoretical Computer Science, vol. 64. Elsevier, 56–72.

Apt, K. R. 2003. *Principles of constraint programming.* Cambridge University Press.

Baeten, J. C., Bergstra, J. A., and Klop, J. W. 1987. Term rewriting systems with priorities. In *2nd International Conference on Rewriting Techniques and Applications*, P. Lescanne, Ed. Lecture Notes in Computer Science, vol. 256. Springer, 83–94.

Bayardo, Jr, R. J. and Miranker, D. P. 1996. Processing queries for first-few answers. In *5th International Conference on Information and Knowledge Management.* ACM, 45–52.

Betz, H. and Frühwirth, T. 2005. A linear-logic semantics for Constraint Handling Rules. In *11th International Conference on Principles and Practice of Constraint Programming*, P. van Beek, Ed. Lecture Notes in Computer Science, vol. 3709. Springer, 137–151.

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, 287–307.

Boyle, J. M., Harmer, T. J., and Winter, V. L. 1997. The TAMPR program transformation system: Design and applications. In *Modern Software Tools for Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhauser Boston Inc., 352–372.

Brant, D. A., Grose, T., Lofaso, B., and Miranker, D. P. 1991. Effects of database size on rule system performance: five case studies. In *17th Inter-*

national Conference on Very Large Data Bases, G. M. Lohman, A. Sernadas, and R. Camps, Eds. Morgan Kaufmann, 287–296.

Brewka, G. and Eiter, T. 1999. Preferred answer sets for extended logic programs. *Artificial Intelligence 109,* 1-2, 297–356.

Browne, J. C., Emerson, E. A., Gouda, M. G., Miranker, D. P., Mok, A. K., Bayardo, Jr, R. J., Chodrow, S. E., Gadbois, D., Haddix, F. F., Hetherington, T. W., Obermeyer, L., Tsou, D.-C., Wang, C.-K., and Wang, R.-H. 1994. A new approach to modularity in rule-based programming. In *6th International Conference on Tools with Artificial Intelligence*. IEEE Computer Society, 18–25.

Bruynooghe, M. 2004. Enhancing a search algorithm to perform intelligent backtracking. *Theory and Practice of Logic Programming 4,* 3, 371–380.

Caferra, R., Echahed, R., and Peltier, N. 2006. Rewriting term-graphs with priority. In *8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, A. Bossi and M. J. Maher, Eds. ACM, 109–120.

Carlsson, M., Ottosson, G., and Carlson, B. 1997. An open-ended finite domain constraint solver. In *9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, H. Glaser, P. H. Hartel, and H. Kuchen, Eds. Lecture Notes in Computer Science, vol. 1292. Springer, 191–206.

Chen, X. and van Beek, P. 2001. Conflict directed backjumping revisited. *Journal of Artificial Intelligence Research 14*, 53–81.

Chin, W.-N., Craciun, F., Khoo, S.-C., and Popeea, C. 2006. A flow-based approach for variant parametric types. In *21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, P. L. Tarr and W. R. Cook, Eds. ACM, 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.

Christiansen, H. 2005. CHR grammars. *Theory and Practice of Logic Programming 5,* 4-5, 467–501.

Christiansen, H. 2006. On the implementation of global abduction. In *7th International Workshop on Computational Logic in Multi-Agent Systems: Revised, Selected and Invited Papers*, K. Inoue, K. Satoh, and F. Toni, Eds. Lecture Notes in Computer Science, vol. 4371. Springer, 226–245.

Christiansen, H. and Dahl, V. 2005. HYPROLOG: A new logic programming language with assumptions and abduction. In *21st International Conference on Logic Programming*, M. Gabrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 159–173.

Christiansen, H. and Have, C. T. 2007. From use cases to UML class diagrams using logic grammars and constraints. In *2007 International Conference on Recent Advances in Natural Language Processing*. 128–132.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. F. 1999. The Maude system. In *10th International Conference on Rewriting Techniques and Applications*, P. Narendran and M. Rusinowitch, Eds. Lecture Notes in Computer Science, vol. 1631. Springer, 240–243.

Coquery, E. and Fages, F. 2003. TCLP: A type checker for CLP($\mathcal{X}$). In *13th Workshop on Logic Programming Environments*, F. Mesnard and A. Serebrenik, Eds. Report CW 371. Department of Computer Science, K.U.Leuven, 17–30.

Coquery, E. and Fages, F. 2005. A type system for CHR. In *2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 421. Department of Computer Science, K.U.Leuven, 19–33.

Dahl, V. 2004. An abductive treatment of long distance dependencies in CHR. In *1st International Workshop on Constraint Solving and Language Processing*, H. Christiansen, P. R. Skadhauge, and J. Villadsen, Eds. Lecture Notes in Computer Science, vol. 3438. Springer, 17–31. Invited Paper.

De Koninck, L. 2007. Mergeable schedules for lazy matching. Tech. Rep. CW 505, Department of Computer Science, K.U.Leuven.

De Koninck, L., Schrijvers, T., and Demoen, B. 2006. INCLP($\mathbb{R}$) - Interval-based nonlinear constraint logic programming over the reals. In *20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. INFSYS Research Report 1843-06-02. TU Wien, 91–100.

De Koninck, L., Schrijvers, T., and Demoen, B. 2007. The correspondence between the Logical Algorithms language and CHR. In *23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 209–223.

De Koninck, L. and Sneyers, J. 2008. Join ordering for CHR: Example programs. http://www.cs.kuleuven.be/~leslie/join/.

Demoen, B. 2002. Dynamic attributes, their hProlog implementation, and a first evaluation. Tech. Rep. CW 350, Department of Computer Science, K.U.Leuven.

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 *2007 ACM Symposium on Applied Computing*, Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, Eds. ACM, 58–64.

Dooms, G., Deville, Y., and Dupont, P. 2005. CP(Graph): Introducing a graph computation domain in constraint programming. In *11th International Conference on Principles and Practice of Constraint Programming*, P. van Beek, Ed. Lecture Notes in Computer Science, vol. 3709. Springer, 211–225.

Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. 2000. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems 22,* 5, 861–931.

Duck, G. J. 2005. Compilation of Constraint Handling Rules. Ph.D. thesis, University of Melbourne.

Duck, G. J. and Schrijvers, T. 2005. Accurate functional dependency analysis for Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 421. Department of Computer Science, K.U.Leuven, 109–124.

Duck, G. J., Stuckey, P. J., and Brand, S. 2006. ACD term rewriting. In *22nd International Conference on Logic Programming*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 117–131.

Duck, G. J., Stuckey, P. J., García de la Banda, M., and Holzbaur, C. 2004. The refined operational semantics of Constraint Handling Rules. In *20th International Conference on Logic Programming*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 90–104.

Duck, G. J., Stuckey, P. J., and Sulzmann, M. 2007. Observable confluence for Constraint Handling Rules. In *23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 224–239.

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence 19,* 1, 17–37.

Fredman, M. L. and Tarjan, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM 34,* 3, 596–615.

Friedman-Hill, E. 2007. *JESS 7.0p2: The rule engine for the Java platform.* http://herzberg.ca.sandia.gov/jess.

Frühwirth, T. 1998. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming 37,* 1-3, 95–138.

Frühwirth, T. 2000. Proving termination of constraint solver programs. In *Joint ERCIM/Compulog Net Workshop on New Trends in Constraints: Selected papers*, K. R. Apt, A. C. Kakas, E. Monfroy, and F. Rossi, Eds. Lecture Notes in Computer Science, vol. 1865. Springer, 298–317.

Frühwirth, T. 2002a. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, D. Fensel, F. Giunchiglia, D. L. 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 *Quantitative Aspects of Programming Languages: Selected Papers*. Electronic Notes in Theoretical Computer Science, vol. 59. Elsevier.

Frühwirth, T. 2005. Parallelizing union-find in Constraint Handling Rules using confluence. In *21st International Conference on Logic Programming*, M. Gabrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 113–127.

Frühwirth, T. 2006. Deriving linear-time algorithms from union-find in CHR. In *3rd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 452. Department of Computer Science, K.U.Leuven, 49–60.

Frühwirth, T., Michel, L., and Schulte, C. 2006. Constraints in procedural and concurrent languages. In *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Foundations of Artificial Intelligence. Elsevier, Chapter 13, 453–494.

Ganzinger, H. and McAllester, D. A. 2001. A new meta-complexity theorem for bottom-up logic programs. In *1st International Joint Conference on Automated Reasoning*, R. Goré, A. Leitsch, and T. Nipkow, Eds. Lecture Notes in Computer Science, vol. 2083. Springer, 514–528.

Ganzinger, H. and McAllester, D. A. 2002. Logical algorithms. In *18th International Conference on Logic Programming*, P. J. Stuckey, Ed. Lecture Notes in Computer Science, vol. 2401. Springer, 209–223.

Garat, D. and Wonsever, D. 2002. A constraint parser for contextual rules. In *22nd International Conference of the Chilean Computer Science Society*. IEEE Computer Society, 234–242.

García, A. J. and Simari, G. R. 2004. Defeasible logic programming: An argumentative approach. *Theory and Practice of Logic Programming 4,* 1-2, 95–138.

Gavanelli, M., Lamma, E., Mello, P., Milano, M., and Torroni, P. 2003. Interpreting abduction in CLP. In *2003 Joint Conference on Declarative Programming, APPIA-GULP-PRODE*, F. Buccafurri, Ed. 25–35.

Giarratano, J. C. 2002. *CLIPS User's Guide, Version 6.20*. http://www.ghg.net/clips/CLIPS.html.

Ginsberg, M. L. 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research 1*, 25–46.

Ginsberg, M. L. and Harvey, W. D. 1992. Iterative broadening. *Artificial Intelligence 55,* 2, 367–383.

Hanus, M. 2006. Adding Constraint Handling Rules to Curry. In *20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. INFSYS Research Report 1843-06-02. TU Wien, 81–90.

Hanus, M. and Steiner, F. 1998. Controlling search in declarative programs. In *10th International Symposium on Programming Language Implementation and Logic Programming / 7th International Conference on Algebraic and Logic Programming*, C. Palamidessi, H. Glaser, and K. Meinke, Eds. Lecture Notes in Computer Science, vol. 1490. Springer, 374–390.

Hart, P. E., Nilsson, N. J., and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4,* 2, 100–107.

Harvey, W. D. and Ginsberg, M. L. 1995. Limited discrepancy search. In *14th International Joint Conference on Artificial Intelligence*. Vol. 1. Morgan Kaufmann, 607–615.

Hermenegildo, M. V., Bueno, F., Cabeza, D., Carro, M., García de la Banda, M. J., López-García, P., and Puebla, G. 1996. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. In *1996 Joint Conference on Declarative Programming, APPIA-GULP-PRODE*, P. Lucio, M. Martelli, and M. Navarro, Eds. 105–110.

Holzbaur, C. 1992. Metastructures versus attributed variables in the context of extensible unification. In *4th International Symposium on Programming Language Implementation and Logic Programming*, M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 631. Springer, 260–268.

Holzbaur, C. and Frühwirth, T. 1998. Constraint Handling Rules reference manual, release 2.2. Tech. Rep. TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence.

Holzbaur, C. and Frühwirth, T. 1999. Compiling Constraint Handling Rules into Prolog with attributed variables. In *1st International Conference on Principles and Practice of Declarative Programming*, G. Nadathur, Ed. Lecture Notes in Computer Science, vol. 1702. Springer, 117–133.

Holzbaur, C. and Frühwirth, T. 2000. A Prolog Constraint Handling Rules compiler and runtime system. *Applied Artificial Intelligence: Special Issue on Constraint Handling Rules 14,* 4, 369–388.

Holzbaur, C., García de la Banda, M., Stuckey, P. J., and Duck, G. J. 2005. Optimizing compilation of Constraint Handling Rules in HAL.

Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules 5, 4 & 5, 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.

Hugo Simões, H. and Florido, M. 2004. TypeTool: A type inference visualization tool. In *13th International Workshop on Functional and (Constraint) Logic Programming*, H. Kuchen, Ed. Technical report AIB-2004-05. Department of Computer Science, RWTH Aachen, 48–61.

Ibaraki, T. and Kameda, T. 1984. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems 9,* 3, 482–502.

ILOG 2001a. *ILOG CPLEX 7.5 reference manual*. ILOG.

ILOG 2001b. *ILOG Solver 5.1: Reference Manual*. ILOG.

Jaffar, J. and Maher, M. J. 1994. Constraint Logic Programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence 27,* 1, 97–109.

Kowalski, R. A. 1979. Algorithm = logic + control. *Communications of the ACM 22,* 7, 424–436.

Krishnamurthy, R., Boral, H., and Zaniolo, C. 1986. Optimization of nonrecursive queries. In *12th International Conference on Very Large Data Bases*, W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds. Morgan Kaufmann, 128–137.

Lam, E. S. and Sulzmann, M. 2007. A concurrent Constraint Handling Rules semantics and its implementation with software transactional memory. In *2007 Workshop on Declarative Aspects of Multicore Programming*. ACM, 19–24.

McAllester, D. A. 1999. On the complexity analysis of static analyses. In *6th International Symposium on Static Analysis*, A. Cortesi and G. Filé, Eds. Lecture Notes in Computer Science, vol. 1694. Springer, 312–329.

Meister, M. 2006. Fine-grained parallel implementation of the preflow-push algorithm in CHR. In *20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. INFSYS Research Report 1843-06-02. TU Wien, 172–181.

Meister, M., Djelloul, K., and Frühwirth, T. 2006. Complexity of a CHR solver for existentially quantified conjunctions of equations over trees. In *11th Annual ERCIM Workshop on Constraint Solving and Constraint Programming*, F. Azevedo, P. Barahona, F. Fages, and F. Rossi, Eds. Lecture Notes in Computer Science, vol. 4651. Springer, 139–153.

Menezes, L., Vitorino, J., and Aurelio, M. 2005. A high performance CHR$^\vee$ execution engine. In *2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 421. Department of Computer Science, K.U.Leuven, 35–45.

Miranker, D. P. 1987. TREAT: A better matching algorithm for AI production system matching. In *6th National Conference on Artificial Intelligence*. AAAI Press, 42–47.

Miranker, D. P., Brant, D. A., Lofaso, B., and Gadbois, D. 1990. On the performance of lazy matching in production systems. In *8th National Conference on Artificial Intelligence*. AAAI Press / The MIT Press, 685–692.

Morawietz, F. 2000. Chart parsing and constraint programming. In *18th International Conference on Computational Linguistics*. Morgan Kaufmann, 551–557.

Morawietz, F. and Blache, P. 2002. Parsing natural languages with CHR. Unpublished Draft.

Müller, H. 2005. Static and dynamic variable sorting strategies for backtracking-based search algorithms. In *19th Workshop on (Constraint) Logic Programming*, A. Wolf, T. Frühwirth, and M. Meister, Eds. Ulmer Informatik-Berichte, vol. 2005-01. Universität Ulm, 99–110.

Penn, G. 2000. Applying Constraint Handling Rules to HPSG. In *1st Workshop on Rule-Based Constraint Reasoning and Programming*.

Pilozzi, P. and De Schreye, D. 2008. Termination analysis of CHR revisited. To appear at the 24th International Conference on Logic Programming.

Pilozzi, P., Schrijvers, T., and De Schreye, D. 2007. Proving termination of CHR in Prolog: A transformational approach. In *9th International Workshop on Termination*, D. Hofbauer and A. Serebrenik, Eds. 30–33.

Proctor, M. 2006. Rete with lazy joins. http://blog.athico.com/2006/11/rete-with-lazy-joins.html.

Proctor, M., Neale, M., Frandsen, M., Griffith, Jr, S., Tirelli, E., Meyer, F., and Verlaenen, K. 2007. *Drools Documentation, Version 4.0.3*. http://www.jboss.com/products/rules.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence 9*, 268–299.

Raiser, F. and Tacchella, P. 2007. On confluence of non-terminating CHR programs. In *4th Workshop on Constraint Handling Rules*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. U.Porto, 63–76.

Ringwelski, G. and Hoche, M. 2005. Impact- and cost-oriented propagator scheduling for faster constraint propagation. In *19th Workshop on (Constraint) Logic Programming*, A. Wolf, T. Frühwirth, and M. Meister, Eds. Ulmer Informatik-Berichte, vol. 2005-01. Universität Ulm, 88–98.

Robin, J., Vitorino, J., and Wolf, A. 2007. Constraint programming architectures: Review and a new proposal. *Journal of Universal Computer Science 13,* 6, 701–720.

Sarna-Starosta, B. and Schrijvers, T. 2008. Transformation-based indexing techniques for Constraint Handling Rules. In *5th Workshop on Constraint Handling Rules*, T. Schrijvers, F. Raiser, and T. Frühwirth, Eds. Number 08-10 in RISC-Linz Report Series. RISC-Linz, 3–17.

Schrijvers, T. 2005. Analyses, optimizations and extensions of Constraint Handling Rules. Ph.D. thesis, K.U.Leuven.

Schrijvers, T. et al. 2008. The Constraint Handling Rules home page. http://www.cs.kuleuven.be/~dtai/projects/CHR/.

Schrijvers, T. and Bruynooghe, M. 2006. Polymorphic algebraic data type reconstruction. In *8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, A. Bossi and M. J. Maher, Eds. ACM, 85–96.

Schrijvers, T. and Demoen, B. 2004. The K.U.Leuven CHR system: implementation and application. In *1st Workshop on Constraint Handling Rules: Selected Contributions*, T. Frühwirth and M. Meister, Eds. Ulmer Informatik-Berichte, vol. 2004-01. Universität Ulm, 1–5.

Schrijvers, T. and Frühwirth, T. 2006. Optimal union-find in Constraint Handling Rules. *Theory and Practice of Logic Programming 6,* 1&2.

Schrijvers, T., Stuckey, P. J., and Duck, G. J. 2005. Abstract interpretation for Constraint Handling Rules. In *7th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, P. Barahona and A. P. Felty, Eds. ACM, 218–229.

Schulte, C. 1999. Comparing trailing and copying for constraint programming. In *1999 International Conference on Logic Programming*, D. De Schreye, Ed. MIT Press, 275–289.

Schulte, C. and Carlsson, M. 2006. Finite domain constraint programming systems. In *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Foundations of Artificial Intelligence. Elsevier, Chapter 14, 495–526.

Schulte, C., Smolka, G., and Würtz, J. 1994. Encapsulated search and constraint programming in Oz. In *2nd International Workshop on Principles and Practice of Constraint Programming*, A. Borning, Ed. Lecture Notes in Computer Science, vol. 874. Springer, 134–150.

Schulte, C. and Stuckey, P. J. 2004. Speeding up constraint propagation. In *10th International Conference on Principles and Practice of Constraint Programming*, M. Wallace, Ed. Lecture Notes in Computer Science, vol. 3258. Springer, 619–633.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. 1979. Access path selection in a relational database management system. In *1979 ACM SIGMOD Conference on Management of Data*, P. A. Bernstein, Ed. ACM, 23–34.

Smith, B. M. and Sturdy, P. 2005. Value ordering for finding all solutions. In *19th International Joint Conference on Artificial Intelligence*, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 311–316.

Sneyers, J., Schrijvers, T., and Demoen, B. 2005. The computational power and complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 421. Department of Computer Science, K.U.Leuven, 3–17.

Sneyers, J., Schrijvers, T., and Demoen, B. 2006a. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. INFSYS Research Report 1843-06-02. TU Wien, 182–191.

Sneyers, J., Schrijvers, T., and Demoen, B. 2006b. Memory reuse for CHR. In *22nd International Conference on Logic Programming*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 72–86.

Sneyers, J., Schrijvers, T., and Demoen, B. 2008. The computational power and complexity of Constraint Handling Rules. To appear in ACM Transactions on Programming Languages and Systems.

Sneyers, J., Van Weert, P., Schrijvers, T., and De Koninck, L. 2008. As time goes by: Constraint Handling Rules — a survey of CHR research from 1998 to 2007. Submitted to Theory and Practice of Logic Programming.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1-3, 17–64.

Steinbrunn, M., Moerkotte, G., and Kemper, A. 1997. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal 6,* 3, 191–208.

Stuckey, P. J. and Sulzmann, M. 2005. A theory of overloading. *ACM Transactions on Programming Languages and Systems 27,* 6, 1216–1269.

Sulzmann, M., Duck, G. J., Peyton-Jones, S., and Stuckey, P. J. 2007. Understanding functional dependencies via Constraint Handling Rules. *Journal of Functional Programming 17,* 1, 83–129.

Sulzmann, M., Schrijvers, T., and Stuckey, P. J. 2006. Principal type inference for GHC-style multi-parameter type classes. In *4th Asian Symposium on Programming Languages and Systems*, N. Kobayashi, Ed. Lecture Notes in Computer Science, vol. 4279. Springer, 26–43.

Sulzmann, M., Wazny, J., and Stuckey, P. J. 2005. Constraint abduction and Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 421. Department of Computer Science, K.U.Leuven, 63–78.

Swami, A. N. and Iyer, B. R. 1993. A polynomial time algorithm for optimizing join queries. In *9th International Conference on Data Engineering*. IEEE Computer Society, 345–354.

Tacchella, P., Meo, M. C., and Gabbrielli, M. 2007. Unfolding in CHR. In *9th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, M. Leuschel and A. Podelski, Eds. ACM, 179–186.

Tarjan, R. E. and van Leeuwen, J. 1984. Worst-case analysis of set union algorithms. *Journal of the ACM 31,* 2, 245–281.

van Hentenryck, P., Perron, L., and Puget, J.-F. 2000. Search and strategies in OPL. *ACM Transactions on Computational Logic 1,* 2, 285–320.

Van Weert, P. 2008. Optimization of CHR propagation rules. To appear at the 24th International Conference on Logic Programming.

Van Weert, P., Schrijvers, T., and Demoen, B. 2005. K.U.Leuven JCHR: A user-friendly, flexible and efficient CHR system for Java. In *2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 421. Department of Computer Science, K.U.Leuven, 47–62.

Van Weert, P., Sneyers, J., and Demoen, B. 2008. Aggregates for CHR through program transformation. In *17th International Symposium on Logic-Based Program Synthesis and Transformation: Revised Selected Papers*, A. King, Ed. Lecture Notes in Computer Science, vol. 4915. 59–73.

Van Weert, P., Sneyers, J., Schrijvers, T., and Demoen, B. 2006. Extending CHR with negation as absence. In *3rd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 452. Department of Computer Science, K.U.Leuven, 125–140.

Van Weert, P., Wuille, P., Schrijvers, T., and Demoen, B. 2008. CHR for imperative host languages. To appear in Lecture Notes in Artificial Intelligence: Special Issue on Recent Advances in Constraint Handling Rules.

Visser, E. 2001. Stratego: A language for program transformation based on rewriting strategies. In *12th International Conference on Rewriting Techniques and Applications*, A. Middeldorp, Ed. Lecture Notes in Computer Science, vol. 2051. Springer, 357–362.

Voets, D., Pilozzi, P., and De Schreye, D. 2007. A new approach to termination analysis of Constraint Handling Rules. In *4th Workshop on Constraint Handling Rules*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. U.Porto, 77–89.

Wallace, M., Novello, S., and Schimpf, J. 1997. ECL^{i}PS^e: A platform for constraint logic programming. *ICL Systems Journal 12,* 1, 159–200.

Walsh, T. 1997. Depth-bounded discrepancy search. In *15th International Joint Conference on Artificial Intelligence*. Vol. 2. Morgan Kaufmann, 1388–1395.

Widom, J. 1996. The Starburst rule system. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 87–109.

Wolf, A. 2000. Toward a rule-based solution of dynamic constraint hierarchies over finite domains. In *1st Workshop on Rule-Based Constraint Reasoning and Programming*.

Wolf, A. 2005. Intelligent search strategies based on adaptive Constraint Handling Rules. *Theory and Practice of Logic Programming 5,* 4-5, 567–594.

Wolf, A., Gruenhagen, T., and Geske, U. 2000. On the incremental adaptation of CHR derivations. *Applied Artificial Intelligence: Special Issue on Constraint Handling Rules 14,* 4, 389–416.

Wolf, A., Robin, J., and Vitorino, J. 2007. Adaptive CHR meets CHR$^\vee$: An extended refined operational semantics for CHR$^\vee$ based on justifications. In *4th Workshop on Constraint Handling Rules*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. U.Porto, 1–15.

Wuille, P., Schrijvers, T., and Demoen, B. 2007. CCHR: The fastest CHR implementation, in C. In *4th Workshop on Constraint Handling Rules*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. U.Porto, 123–137.

List of Symbols

The following symbols are used frequently throughout the text. The page numbers refer to the first occurrence of the symbol in question.

Symbol	Description	Page
$\mathsf{vars}(A)$	the variables occurring in A	10
$\bar{\exists}_A F$	$\exists x_1, \ldots, \exists x_n F$ with $x_1, \ldots, x_n = \mathsf{vars}(F) \setminus \mathsf{vars}(A)$	10
$\forall_A F$	$\forall x_1, \ldots, \forall x_n F$ with $x_1, \ldots, x_n = \mathsf{vars}(A)$	10
ϵ	empty sequence	7
$+\!\!+$	sequence concatenation	7
$[e_1, \ldots, e_n]$	sequence of elements $e_1, \ldots, e_n$	7
$[H \mid T]$	$[H] +\!\!+ T$	7
$\uplus$	multi-set union or disjoint union	11
$\mathcal{D}$	built-in constraint theory	11
$\theta(r)$	instance of rule r with matching substitution θ	11
$\mathrm{CHR}^{\vee}$	CHR with disjunctive rule bodies	72
$\mathrm{CHR}^{\mathrm{rp}}$	CHR with rule priorities	27
$\mathrm{CHR}^{\mathrm{rp}}_{\vee}$	the combination of $\mathrm{CHR}^{\vee}$ and $\mathrm{CHR}^{\mathrm{rp}}$	75
$\mathrm{CHR}^{\mathrm{brp}}_{\vee}$	$\mathrm{CHR}^{\vee}$ with branch and rule priorities	75
ω_t	theoretical operational semantics of CHR	11
ω_r	refined operational semantics of CHR	14
$\omega_t^{\vee}$	theoretical operational semantics of $\mathrm{CHR}^{\vee}$	72
$\omega_r^{\vee\star}$	extended & refined operational semantics of $\mathrm{CHR}^{\vee}$	91
ω_p	priority semantics of $\mathrm{CHR}^{\mathrm{rp}}$	36
ω_{rp}	refined priority semantics of $\mathrm{CHR}^{\mathrm{rp}}$	54
$\omega_p^{\vee}$	priority semantics of $\mathrm{CHR}^{\mathrm{brp}}_{\vee}$	76
$\omega_p^{\vee\star}$	extended priority semantics of $\mathrm{CHR}^{\mathrm{brp}}_{\vee}$	92

Symbol	Description	Page
$\overset{\omega}{\longmapsto}_P$	transition between two states in program P under operational semantics ω	11
$\overset{\omega}{\longmapsto}^*_P$	derivation between two states in program P under operational semantics ω	12
$\overset{\omega}{\not\longmapsto}_P$	final state in program P under operational semantics ω	12
$c\#i$	constraint c with identifier i	11
$c\#i : j$	identified constraint $c\#i$ being matched with the j^{th} occurrence of its predicate symbol	15
$c\#i : j \ @ \ p$	identified constraint $c\#i$ being matched with the j^{th} occurrence of priority p of its predicate symbol	54
$\mathsf{chr}(c\#i)$	CHR constraint c	11
$\mathsf{id}(c\#i)$	CHR constraint identifier i	11
$\mathsf{just}(c^J)$	justification J	92
$\mathcal{BP}$	domain of branch priorities	76
bp_0	initial branch priority	76
$\preceq$	order relation over branch priorities	76
$\mathcal{J}^k_\Theta$	partial join of k heads following join order Θ	157
C^k_Θ	cost of joining k heads following join order Θ	158
$\sigma_{\mathsf{eq}}(k)$	selectivity of equality guard after joining k heads	157
$\sigma_\star(k)$	selectivity of remaining guard after joining k heads	157
$\mu(k)$	multiplicity of lookups for which the equality guard is satisfied, after joining k heads	157

List of Publications

Articles in international reviewed journals

De Koninck, L., Schrijvers, T., and Demoen, B. 2008. A flexible search framework for CHR. To appear in Lecture Notes in Artificial Intelligence: Special Issue on Recent Advances in Constraint Handling Rules.

Contributions at international conferences, published in proceedings

Duck, G. J., De Koninck, L., and Stuckey, P. J. 2008. Cadmium: An implementation of ACD term rewriting. To appear at the 24th International Conference on Logic Programming.

De Koninck, L., Stuckey, P. J., and Duck, G. J. 2008. Optimizing compilation of CHR with rule priorities. In *9th International Symposium on Functional and Logic Programming*, J. Garrigue and M. Hermenegildo, Eds. Lecture Notes in Computer Science, vol. 4989. Springer, 32–47.

De Koninck, L., Schrijvers, T., and Demoen, B. 2007. The correspondence between the Logical Algorithms language and CHR. In *23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 209–223.

De Koninck, L., Schrijvers, T., and Demoen, B. 2007. User-definable rule priorities for CHR. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, M. Leuschel and A. Podelski, Eds. ACM Press, 25–36.

Contributions at international workshops, published in proceedings

De Koninck, L. and Sneyers, J. 2007. Join ordering for Constraint Handling Rules. In *4th Workshop on Constraint Handling Rules*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. U.Porto, 107–121.

De Koninck, L., Schrijvers, T., and Demoen, B. 2006. Search strategies in CHR(Prolog). In *3rd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Report CW 452. Department of Computer Science, K.U.Leuven, 109–123.

De Koninck, L., Schrijvers, T., and Demoen, B. 2006. INCLP($\mathbb{R}$) - Interval-based nonlinear constraint logic programming over the reals. In *20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. INFSYS Research Report 1843-06-02. TU Wien, 91–100.

Technical Reports

De Koninck, L. 2007. Mergeable schedules for lazy matching. Tech. Rep. CW 505, Department of Computer Science, K.U.Leuven.

De Koninck, L., Stuckey, P. J., and Duck, G. J. 2007. Optimized compilation of CHR$^{\text{rp}}$. Tech. Rep. CW 499, Department of Computer Science, K.U.Leuven.

De Koninck, L., Schrijvers, T., and Demoen, B. 2007. The correspondence between the Logical Algorithms language and CHR. Tech. Rep. CW 480, Department of Computer Science, K.U.Leuven.

De Koninck, L., Schrijvers, T., and Demoen, B. 2007. CHR$^{\text{rp}}$: Constraint Handling Rules with rule priorities. Tech. Rep. CW 479, Department of Computer Science, K.U.Leuven.

De Koninck, L., Schrijvers, T., and Demoen, B. 2006. Flexible search strategies in Prolog CHR. Tech. Rep. CW 447, Department of Computer Science, K.U.Leuven.

Biography

Leslie De Koninck was born on the 5th of July 1983 in Turnhout, Belgium. After graduating from high school at the Sint-Jan Berchmanscollege (Westmalle) in 2001, he started studying computer science at K.U.Leuven. In 2005, he graduated summa cum laude and received the degree of Master in Computer Science (Dutch: Licentiaat in de Informatica). His Masters thesis was entitled "Constraint Solvers for SWI-Prolog" and was supervised by Prof. Bart Demoen.

In August 2005, he joined the Declarative Languages and Artificial Intelligence (DTAI) research group at K.U.Leuven as a PhD student under supervision of Prof. Demoen. Until the end of 2005, he was employed as scientific collaborator and since 2006 he is supported by a PhD grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). He was a visiting scholar at Melbourne University in June 2007 and February and March 2008 under supervision of Prof. Peter Stuckey.

Uitvoeringscontrole voor Constraint Handling Rules

Nederlandstalige Samenvatting

1 Inleiding

Uitvoeringscontrole voor CHR

Constraint Programming (CP) (Apt 2003) is een hoog-niveau declaratief programmeerparadigma waarbij problemen gemodelleerd worden door middel van *constraints* (beperkingen), t.t.z., relaties tussen de probleemvariabelen waaraan alle oplossingen moeten voldoen. Vele praktisch relevante problemen kunnen eenvoudig beschreven worden via constraints. Zo bestaan er toepassingen in onder meer productieplanning, het opstellen van dienstroosters voor personeel, het ontwerp en de controle van elektrische circuits, en de verwerking van natuurlijke taal. Een constraint programming systeem omvat een *constraintoplosser*. Deze heeft als taak waarderingen

voor de probleemvariabelen te vinden die aan alle constraints voldoen. Constraint Programming systemen worden traditioneel opgedeeld op basis van het domein van de variabelen en het type van de ondersteunde constraints. Voorbeelden zijn constraints over eindige domeinen en lineaire constraints over de reële getallen. Vele praktische problemen zijn echter moeilijk te modelleren in termen van de standaard constraintdomeinen. Daarom zijn er langs de ene kant systemen ontwikkeld voor meer gespecialiseerde domeinen zoals constraints over eindige verzamelingen (Dovier et al. 2000) en constraints over grafen (Dooms et al. 2005). Langs de andere kant zijn er faciliteiten ontwikkeld die het eenvoudiger maken om gespecialiseerde constraintoplossers te implementeren. In de context van Constraint Logic Programming (CLP) (Jaffar and Maher 1994), een combinatie van Constraint Programming en logisch programmeren, zijn dit onder meer indexicals (Carlsson et al. 1997), geattribueerde variabelen (Holzbaur 1992), en in het bijzonder Constraint Handling Rules (Frühwirth 1998).

Constraint Handling Rules (CHR) is een regelgebaseerde taal die speciaal ontworpen is voor de implementatie van gespecialiseerde constraintoplossers. Taalconcepten als meerhoofdige regels, wachters en multi-set semantiek maken CHR een zeer flexibele taal voor de specificatie van de logica van een constraintoplosser. Flexibele uitvoeringscontrole ontbreekt echter bijna volledig. Uitvoeringscontrole is van fundamenteel belang voor de efficiëntie van C(L)P systemen, zie bv. (Schulte and Stuckey 2004; Ringwelski and Hoche 2005). Tot dusver is hier echter binnen de context van CHR nauwelijks aandacht aan besteed. In dit proefschrift presenteren we een oplossing voor het probleem van uitvoeringscontrole in CHR. In het

bijzonder breiden we CHR uit met hoog-niveau faciliteiten om de specificatie van uitvoeringsstrategieën te ondersteunen. We bieden voldoende flexibiliteit aan om het soort uitvoeringscontrole toe te laten dat nodig is voor het efficiënt oplossen van constraints. Bovendien beperken we het aantal nieuwe taalconcepten die de programmeur moet leren, evenals het meerverbruik in looptijd (runtime overhead) dat resulteert uit een meer flexibel uitvoeringsmechanisme.

Doelstelling en Bijdragen

De hoofddoelstelling van deze thesis is het ontwerp, de implementatie en de analyse van taalconcepten en -constructies die een flexibele, efficiënte, en hoog-niveau vorm van uitvoeringscontrole toelaten in Constraint Handling Rules. Het oplossen van constraints bestaat in het algemeen zowel uit *propagatie* als uit *zoeken*. Tijdens de propagatie wordt informatie van verschillende constraints gecombineerd, hetgeen kan leiden tot de vereenvoudiging van constraints en de verwijdering van inconsistente waarden uit de domeinen van de probleemvariabelen. Tijdens het zoeken worden speculatief waarden toegekend aan de variabelen. De opdeling van het oplossen van constraints in propagatie en zoeken leidt ook tot een overeenkomstige opdeling van de uitvoeringsstrategie in een propagatiestrategie en een zoekstrategie.

We geven nu een overzicht van onze belangrijkste bijdragen. Onze eerste twee bijdragen bestaan in het ontwerp en de implementatie van taaluitbreidingen voor CHR om de specificatie van propagatie- en zoekstrategieën te ondersteunen:

- In Hoofdstuk 3 breiden we CHR uit met gebruikersgedefinieerde regelprioriteiten, tot CHR^{rp}. CHR-regels komen overeen met constraint propagatoren: ze worden gebruikt om combinaties van constraints te vereenvoudigen, alsook om impliciete informatie expliciet te maken in de vorm van (logisch redundante) constraints. Daarom laten regelprioriteiten dan ook toe om de propagatiestrategie te specifiëren. We tonen hoe CHR^{rp} kan gebruikt worden om beknopt strategieën uit te drukken die leiden tot efficiëntere programma's. We presenteren een geoptimaliseerde implementatie van CHR^{rp} en evalueren deze empirisch.

- Hoofdstuk 4 breidt het werk in het vorige hoofdstuk uit met faciliteiten voor de controle van de zoekstrategie. Gewone CHR biedt geen ondersteuning voor zoeken, maar de taaluitbreiding $\text{CHR}^{\vee}$ doet dit wel. Onze aanpak combineert $\text{CHR}^{\vee}$ en CHR^{rp} in een nieuwe taal genaamd $\text{CHR}^{\text{brp}}_{\vee}$. Hierin wordt de propagatiestrategie bepaald door middel van regelprioriteiten, en de zoekstrategie door middel van vertakkingsprioriteiten (branch priorities). We tonen hoe verscheidene gebruikelijke zoekstrategieën gespecifieerd kunnen worden in $\text{CHR}^{\text{brp}}_{\vee}$, en hoe zulke strategieën gecombineerd kunnen worden. We presenteren ook een uitbreiding van $\text{CHR}^{\text{brp}}_{\vee}$ om conflict-gerichte backjumping te ondersteunen.

Onze overige bijdragen zijn de volgende:

- CHR$^{\text{rp}}$ is geïnspireerd door een gerelateerde bottom-up logische programmeertaal, gepresenteerd in (Ganzinger and McAllester 2002). In dat werk vormt de *Logical Algorithms*[1] taal de basis voor een meta-complexiteitsresultaat voor logische programmeertalen. Dit resultaat geeft de tijdscomplexiteit van een logisch programma in termen van het aantal regelvuringen en andere concepten op het niveau van de taal (t.t.z., zonder een analyse van laagniveau implementatiedetails te vereisen). In Hoofdstuk 5 stellen we formeel de overeenkomst tussen CHR$^{\text{rp}}$ en Logical Algorithms vast, en presenteren we een uitbreiding van het Logical Algorithms meta-complexiteitsresultaat. Dit laat ons toe om op een hoog-niveau manier de tijdscomplexiteit van CHR$^{\text{rp}}$-programma's te analyseren.
- De optimalisatie van de *join order* (verbindingsvolgorde) is een belangrijk deel van de geoptimaliseerde compilatie van CHR-programma's. Deze bestaat erin een optimale volgorde te vinden om de hoofden van een CHR-regel te matchen met constraints uit de constraint store. In Hoofdstuk 6 presenteren we de eerste formele en diepgaande studie van het join order optimalisatieprobleem. In het bijzonder stellen we een kostenmodel voor het matchen van meerhoofdige regels voor. We presenteren benaderingen voor de parameters van dit model, en tonen hoe we de kost kunnen minimaliseren binnen het model. Dit werk is van toepassing op CHR in het algemeen; een verfijning van het kostenmodel voor het specifieke geval van CHR$^{\text{rp}}$ wordt ook gegeven.

Hoofdstuk 2 geeft de nodige achtergrond i.v.m. CHR en Hoofdstuk 7 presenteert een algemeen besluit voor dit proefschrift.

2 Achtergrond: Constraint Handling Rules

Dit hoofdstuk introduceert achtergrondinformatie betreffende de Constraint Handling Rules taal. We veronderstellen dat de lezer enigszins vertrouwd is met logisch programmeren en Prolog, maar veronderstellen geen kennis van CHR.

Constraint Handling Rules

Constraint Handling Rules (CHR) (Frühwirth 1998) is een hoog-niveau regelgebaseerde programmeertaal, ontworpen om de implementatie van applicatie-specifieke constraintoplossers te vergemakkelijken. CHR regels zijn meerhoofdige, conditionele (*guarded*) herschrijfregels die opereren op een multi-set van gebruikersgedefinieerde constraints. Deze multi-set wordt de *CHR constraint store* genoemd. Een

[1] De taal heeft geen naam gekregen in (Ganzinger and McAllester 2002), maar de paper zelf heet "Logical Algorithms".

CHR-constraintoplosser draait bovenop een gastheer-taal, dewelke een ingebouwde constraintoplosser aanbiedt.

CHR-regels implementeren constraint simplificatie en propagatie. Ze laten toe om gebruikersgedefinieerde constraints te beschrijven in termen van lager-niveau ingebouwde constraints die vervolgens opgelost worden door de gastheer-taal. Zo kunnen implementaties van CHR met Prolog als gastheer-taal gebruik maken van het Prolog unificatie-algoritme om gelijkheidsconstraints over Herbrand-termen op te lossen.

Syntax

Een constraint $c(t_1, \ldots, t_n)$ is een atoom van predicaat c/n met t_i een waarde in de gastheer-taal (bv. een Herbrand-term in Prolog) voor $1 \leq i \leq n$. Er zijn twee types van constraints: ingebouwde constraints en CHR-constraints (ook wel gebruikersgedefinieerde constraints genoemd). De CHR-constraints worden opgelost door het CHR-programma, en de ingebouwde constraints door een onderliggende oplosser (bv. het Prolog unificatie-algoritme). Er zijn drie types van Constraint Handling Rules: *simplificatieregels*, *propagatieregels* en *simpagatieregels*. Zij hebben de volgende vorm:

$$\begin{array}{lllll} \textbf{Simplificatie} & r \ @ & H^r & \iff & g \mid B \\ \textbf{Propagatie} & r \ @ \ H^k & & \implies & g \mid B \\ \textbf{Simpagatie} & r \ @ \ H^k \setminus H^r & & \iff & g \mid B \end{array}$$

waarbij r de *naam* van de regel is, H^k en H^r zijn sequenties van CHR-constraints en worden de *hoofden* van de regel genoemd, de *wachter* van de regel, g, is een conjunctie van ingebouwde constraints, en het *lichaam* van de regel, B, is een multi-set van zowel CHR- als ingebouwde constraints. Doorheen de tekst gebruiken we de simpagatieregel als voorbeeld voor eender welk type van regel. Een programma P is een verzameling van CHR-regels.

Operationele Semantiek

Operationeel hebben CHR-constraints een multi-set semantiek. Om het onderscheid te kunnen maken tussen verschillende voorkomens van syntactisch gelijke constraints, krijgen CHR-constraints een uniek identificatienummer.

De theoretische operationele semantiek van CHR, ω_t, is in (Duck et al. 2004) gegeven als een toestandsmachine. Een CHR-uitvoeringstoestand σ wordt voorgesteld als een tupel $\langle G, S, B, T \rangle_n$ waarbij G het *doel* (goal) is, t.t.z., een multi-set van constraints die opgelost moeten worden; S is de CHR constraint store; B is de toestand van de ingebouwde constraintoplosser; T is de propagatiegeschiedenis, die bijhoudt welke regelinstanties reeds gevuurd hebben; en n is het eerstvolgende identificatienummer dat gebruikt kan worden voor een CHR-constraint.

De ω_t semantiek definieert drie transities: de **Solve** transitie lost een ingebouwde constraint uit het doel op, de **Introduce** transitie verplaatst een CHR-constraint van het doel naar de CHR constraint store, en de **Apply** transitie vuurt een regelinstantie. Een regelinstantie $\theta(r)$ instantieert een regel met CHR-constraints die overeenkomen met de hoofden, gebruikmakend van de matching-substitutie θ. De propagatiegeschiedenis voorkomt dat een regelinstantie meermaals vuurt.

Een CHR-derivatie start vanuit een initiële toestand $\langle G, \emptyset, \texttt{true}, \emptyset \rangle_1$ met G de constraints die opgelost moeten worden. Een toestand wordt *finaal* genoemd als er geen transities meer op van toepassing zijn. Als de toestand van de ingebouwde oplosser consistent is, wordt de toestand *succesvol* genoemd; zoniet spreken we van een *gefaalde* finale uitvoeringstoestand.

De *verfijnde operationele semantiek* van CHR, ω_r genoteerd, is in (Duck et al. 2004) geïntroduceerd als een formalisatie van het uitvoeringsmechanisme van de meeste actuele CHR-implementaties. Hoewel initieel bedoeld als beschrijvend, is de semantiek verworden tot een voorschrift voor hoe CHR-implementaties moeten werken. De reden is dat de ω_t semantiek te niet-deterministisch is, en vele programma's dan ook geschreven zijn met de ω_r semantiek in het achterhoofd. Deze programma's werken dikwijls niet correct onder de ω_t semantiek (t.t.z., ze zijn niet *confluent*). In Hoofdstuk 3 geven we een hoog-niveau alternatief voor de ω_r semantiek, dat toch nog aanzienlijk veel uitvoeringscontrole toelaat.

Programma-Eigenschappen: Confluentie

In (Abdennadher 1997) wordt een criterium gegeven om te beslissen of een (terminerend) CHR-programma confluent is onder de ω_t semantiek. Intuïtief betekent confluentie dat de antwoorden voor een gegeven doel, onafhankelijk zijn van het uitvoeringspad dat gevolgd is. In het bijzonder willen we dat wanneer er in een gegeven toestand meerdere transities mogelijk zijn, de resulterende toestanden uiteindelijk terug samengevoegd kunnen worden. Onder de verfijnde operationele semantiek volstaat de test die voor ω_t gebruikt wordt niet. In (Duck 2005, Hoofdstuk 6) wordt een praktische confluentietest voor ω_r gegeven. Deze test is echter niet volledig.

3 CHR met Regelprioriteiten

Dit hoofdstuk introduceert CHR$^{\text{rp}}$: Constraint Handling Rules met regelprioriteiten. CHR$^{\text{rp}}$ biedt flexibele controle van de propagatiestrategie aan, hetgeen ontbreekt in CHR. We geven een formele operationele semantiek voor de uitgebreide taal en tonen aan dat deze een instantie is van de theoretische operationele semantiek van CHR. We relateren regelprioriteiten aan alternatieve vormen van uitvoeringscontrole. Verder tonen we aan dat CHR$^{\text{rp}}$ efficiënt gecompileerd kan

worden naar de onderliggende gastheer-taal. Om dit te bereiken introduceren we verscheidene optimalisaties die empirisch geëvalueerd worden. De CHR^{rp}-compiler wordt ook vergeleken met het state-of-the-art K.U.Leuven CHR-systeem. Dit vergelijk toont aan dat we een gelijkaardige performantie kunnen aanbieden en tegelijkertijd een veel hoger niveau van uitvoeringscontrole.

Motivatie

In deze sectie geven we ter motivatie enkele potentiële toepassingsgebieden voor de voorgestelde taaluitbreiding.

Constraint Propagatoren Constraintoplossers maken in het algemeen gebruik van constraint propagatoren om inconsistente waarden weg te filteren uit de domeinen van de constraint variabelen. Efficiënte oplossers maken gebruik van een prioriteitenschema om ervoor te zorgen dat propagatoren die computationeel goedkoper zijn, of een grotere verwachte impact hebben, eerder uitgevoerd worden (Ringwelski and Hoche 2005; Schulte and Stuckey 2004). CHR-regels worden vaak gebruikt als templates voor constraint propagatoren die dan geïnstantieerd worden met concrete constraints.

Zachte Constraints In (Bistarelli et al. 2004) wordt een raamwerk voorgesteld om om te gaan met *zachte constraints* in CHR, gebaseerd op het *c-semiring* formalisme. In dit raamwerk worden scores toegekend aan elke mogelijke waarde voor de probleemvariabelen. Harde constraints kunnen geïmplementeerd worden door een nul score toe te kennen aan elke waarde die niet aan de constraint voldoet. Deze waarden moeten dan verder niet meer beschouwd worden. Het is nuttig om de harde constraints eerst te verwerken omdat zij waarden uit de domeinen van de variabelen verwijderen, terwijl zachte constraints in het algemeen enkel de score van deze waarden wijzigen. Bovendien vereist constraint propagatie voor zachte constraints de combinatie van de mogelijke waarden voor de verschillende probleemvariabelen, hetgeen computationeel duur kan zijn.

In CHR^{rp} kunnen we een hoge prioriteit toekennen aan regels die harde constraints implementeren, een gemiddelde prioriteit aan regels die zachte constraints implementeren, en een lage prioriteit aan regels die speculatief waarden toekennen aan variabelen (zoeken). Bovendien kunnen we verder differentiëren tussen goedkopere en duurdere zachte constraints, bv. afhankelijk van het aantal variabelen waarop de constraint betrekking heeft.

Constraint Store Invarianten Het is vaak wenselijk om bepaalde representatie-invarianten op te leggen aan de constraints in de CHR constraint store. Een voorbeeld van zulke invarianten is set semantiek: de constraint store bevat geen syntactisch gelijke constraints. Dit soort invarianten kan geschonden

worden wanneer we een nieuwe constraint opleggen. We kunnen echter speciale CHR-regels voorzien die deze invarianten herstellen. Zulke regels moeten dan wel vuren voor eender welke andere regel die verwacht dat de invarianten gelden.

Dynamische Regelprioriteiten Regelprioriteiten worden *dynamisch* genoemd als ze afhangen van de argumenten van de constraints die een regelinstantie vormen. Dynamische regelprioriteiten zijn slechts gekend tijdens de uitvoering en verschillende instanties van dezelfde regel kunnen een verschillende prioriteit hebben. We hebben dynamische regelprioriteiten succesvol toegepast voor de implementatie van Dijkstra's kortstepadalgoritme en een oplosser voor Sudoku puzzels.

CHR$^{\text{rp}}$: CHR met Regelprioriteiten

CHR$^{\text{rp}}$ breidt CHR uit met gebruikersgedefinieerde regelprioriteiten. In deze sectie introduceren we de syntax en semantiek van CHR$^{\text{rp}}$.

De syntax van CHR$^{\text{rp}}$ is compatibel met die van gewone CHR. Een CHR$^{\text{rp}}$-simpagatieregel ziet er als volgt uit:

$$p :: r \ @ \ H^k \setminus H^r \iff g \mid B$$

waarbij r, H^k, H^r, g en B zijn als voordien, en de *regelprioriteit* p een aritmetische uitdrukking is waarvan de variabelen ook voorkomen in de regelhoofden. Een regel wiens prioriteit geen variabelen bevat, wordt een regel met *statische* prioriteit genoemd; zoniet is het een regel met *dynamische* prioriteit.

De operationele semantiek van CHR$^{\text{rp}}$ wordt de prioriteitensemantiek genoemd, en genoteerd als ω_p. Deze ω_p semantiek is een verfijning van de ω_t semantiek van CHR met een minimale hoeveelheid determinisme om zo regelprioriteiten te ondersteunen. In het bijzonder beperkt de ω_p semantiek de toepasbaarheid van de **Apply** transitie: deze is enkel toepasbaar als het doel leeg is, en een regelinstantie kan enkel vuren als er geen toepasbare regelinstantie met een hogere prioriteit bestaat.

Programma-Eigenschappen: Confluentie

Een programma dat niet confluent is onder de ω_t semantiek is dat mogelijk wel onder de ω_p semantiek. Wanneer we echter de confluentietest voor ω_t proberen aan te passen aan ω_p, krijgen we te maken met gelijkaardige problemen als bij confluentie onder de verfijnde operationele semantiek ω_r. Een aantal concepten uit de praktische confluentietest voor ω_r, beschreven in (Duck 2005, Hoofdstuk 6), zijn eveneens van toepassing op de ω_p semantiek van CHR$^{\text{rp}}$. Een praktische confluentietest voor ω_p blijft echter toekomstig werk.

We kunnen confluentie voor ω_p relateren aan observeerbare confluentie zoals beschreven in (Duck et al. 2007). Observeerbare confluentie is confluentie voor toestanden die bereikbaar zijn vanuit een geldige initiële toestand. Deze bereikbare toestanden voldoen meestal aan een aantal invarianten. We kunnen deze invarianten in bepaalde gevallen formuleren als CHR$^{\text{rp}}$-regels met een hoge prioriteit. Deze regels laten toestanden die niet aan de invarianten voldoen, deterministisch falen. Confluentie van het resulterende CHR$^{\text{rp}}$-programma onder de ω_p semantiek is dan equivalent met observeerbare confluentie onder de ω_t semantiek.

Compilatie van CHR$^{\text{rp}}$

De compilatie van CHR$^{\text{rp}}$ wordt beschreven aan de hand van een verfijning van de operationele semantiek van CHR$^{\text{rp}}$. Deze verfijning, genoteerd als ω_{rp}, is een combinatie van de ω_p semantiek van CHR$^{\text{rp}}$ en de ω_r semantiek van gewone CHR. De compilatie is dan ook gebaseerd op gelijkaardige principes als de compilatie van gewone CHR, maar dan aangepast om prioriteiten te ondersteunen. De ω_{rp} semantiek vereist dat elke constraint de prioriteit kent van de regelinstanties waarin deze participeert. Voor regels met een dynamische prioriteit geldt dit in het algemeen niet. Om dit probleem op te lossen maken we gebruik van een broncodetransformatie die zorgt dat de resulterende code wel aan de vereisten voldoet.

Het basiscompilatieschema biedt nog vele mogelijkheden tot optimalisatie. Zo maakt het veelvuldig gebruik van een prioriteitsrij om taken te schedulen. Eén reeks optimalisaties bestaat erin het aantal operaties op deze prioriteitsrij te reduceren. Een andere optimalisatie heeft betrekking op indexering. Constraints worden geïndexeerd op (combinaties van) hun argumenten om zo snel de nodige constraints te kunnen opzoeken tijdens het matchen. Dit indexeren is echter een vrij kostelijke operatie, en het loont dan ook de moeite om de indexering (gedeeltelijk) uit te stellen. Een laatste optimalisatie betreft een reductie van het aantal regels dat moet bekeken worden wanneer een nieuwe constraint wordt toegevoegd.

Evaluatie

We hebben onze CHR$^{\text{rp}}$-implementatie geëvalueerd op een aantal benchmarks. De evaluatie toont aan dat de optimalisaties t.o.v. het basiscompilatieschema doeltreffend zijn, en dat ons systeem reeds competitief is t.o.v. het state-of-the-art K.U.Leuven CHR-systeem (op programma's met een gelijkaardige uitvoering) terwijl het een veel hoger niveau aan uitvoeringscontrole aanbiedt.

4 Een Flexibel Zoekraamwerk bovenop CHR$^{\text{rp}}$

Dit hoofdstuk introduceert een raamwerk voor de specificatie van (boom-)zoekstrategieën in CHR met disjunctie (CHR$^{\vee}$). We ondersteunen de specificatie van ge-

bruikelijke zoekstrategieën zoals diepte-eerst, breedte-eerst en beste-eerst, evenals constrained optimalisatie d.m.v. branch & bound zoeken. Het raamwerk is gegeven als een uitbreiding van CHR met regelprioriteiten (CHR^{rp}) waarin elke tak van de zoekboom een *vertakkingsprioriteit* wordt toegekend. Deze aanpak leidt tot een uniforme oplossing voor uitvoeringscontrole in CHR.

Inleiding

CHR beoogt een hoog-niveau taal voor de implementatie van constraintoplossers te zijn, en inderdaad, ze is zeer geschikt om de propagatielogica van constraintoplossers te representeren: de notie van een constraint propagator komt overeen met een CHR-regel. De taaluitbreiding $\text{CHR}^{\vee}$ (Abdennadher and Schütz 1998) geeft ons ook een hoog-niveau manier om het zoekaspect van het oplossen van constraints uit te drukken. Echter, omwille van de niet-deterministische operationele semantiek van deze aspecten, blijft een CHR-programma slechts een abstractie van een constraintoplosser. De meeste constraint problemen zijn namelijk gewoonweg te groot om toevertrouwd te worden aan een niet-deterministische oplosser. Een geschikte constraintoplossingsstrategie kan echter de computationele kost verminderen met meerdere grootte-ordes, en dus het verschil maken tussen een onhaalbare en een haalbare aanpak.

Het is om deze reden dat de meeste state-of-the-art constraintoplossers mogelijkheden aanbieden om de gewenste oplossingsstrategie te selecteren en/of te specifiëren. De oplossingsstrategie bestaat doorgaans uit twee aspecten: propagatieprioriteiten voor conjuncties en zoekprioriteiten voor disjuncties. Propagatieprioriteiten kunnen gespecifieerd worden m.b.v. regelprioriteiten: deze hebben we bestudeerd in Hoofdstuk 3. Dit hoofdstuk behandelt de zoekprioriteiten:

1. We presenteren $\text{CHR}_{\vee}^{\text{brp}}$, een hoog-niveau aanpak om de uitvoeringscontrole in $\text{CHR}^{\vee}$ te specifiëren. $\text{CHR}_{\vee}^{\text{brp}}$ breidt $\text{CHR}^{\vee}$ uit met zowel vertakkings- als regelprioriteiten.

2. We tonen hoe standaard boomzoekstrategieën zoals diepte-eerst, breedte-eerst, diepte-eerst iteratief verdiepen, en beperkte-discrepantie zoeken in $\text{CHR}_{\vee}^{\text{brp}}$ uitgedrukt kunnen worden.

3. We tonen hoe conflict-gerichte backjumping gerealiseerd kan worden door ons raamwerk uit te breiden met *justifications* (verantwoordingen). Ons werk breidt (Wolf et al. 2007) uit door de zoekstrategie niet te beperken tot links-naar-rechts diepte-eerst, en door correctheid en optimaliteit te behandelen.

CHR met Disjunctie

Constraint Handling Rules werd uitgebreid met disjuncties in de regellichamen in (Abdennadher and Schütz 1998). De resulterende taal wordt genoteerd als $\text{CHR}^{\vee}$.

De syntax van $\text{CHR}^\vee$ is dezelfde als die van gewone CHR, behalve dan dat regellichamen in $\text{CHR}^\vee$ formules zijn, opgebouwd uit atomen die gecombineerd worden met conjuncties en disjuncties. We definiëren een operationele semantiek $\omega_t^\vee$ voor $\text{CHR}^\vee$, als een uitbreiding van de ω_t semantiek van CHR. Een $\omega_t^\vee$ uitvoeringstoestand is een multi-set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ van ω_t uitvoeringstoestanden. Elk element $\sigma_i \in \Sigma$ stelt een alternatieve oplossing voor.

Een Combinatie van CHR^{rp} en $\text{CHR}^\vee$

In deze sectie combineren we CHR^{rp} en $\text{CHR}^\vee$ tot een flexibel raamwerk voor de specificatie van zowel de zoek- als de propagatiestrategie die de CHR-constraintoplosser dient te gebruiken.

$\text{CHR}_\vee^{\text{brp}}$ combineert $\text{CHR}^\vee$ en CHR^{rp} en breidt deze uit met vertakkingsprioriteiten om de specificatie van zoekstrategieën te ondersteunen. Een $\text{CHR}_\vee^{\text{brp}}$-simpagatieregel ziet er als volgt uit:

$$(bp, rp) :: r \ @ \ H^k \setminus H^r \iff g \mid bp_1 :: B_1 \vee \ldots \vee bp_m :: B_m$$

Hierbij zijn r, H^k, H^r en g als eerder gedefinieerd. De regelprioriteit rp is als in CHR^{rp}. De vertakkingsprioriteit bp is een term die tijdens de uitvoering gematcht wordt met de prioriteit van het alternatief onder beschouwing. Ze laat toe om gebruik te maken van deze prioriteit in de bepaling van de prioriteiten van nieuw gecreëerde alternatieven.

Het regellichaam is een verzameling van disjuncten B_i, die allen geannoteerd zijn met een vertakkingsprioriteit bp_i $(1 \leq i \leq m)$. Een $\text{CHR}_\vee^{\text{brp}}$-programma is een tupel $\langle \mathcal{R}, \mathcal{BP}, bp_0, \preceq \rangle$ waarbij $\mathcal{R}$ een verzameling van $\text{CHR}_\vee^{\text{brp}}$-*regels* is, $\mathcal{BP}$ is het *domein* van de vertakkingsprioriteiten, $bp_0 \in \mathcal{BP}$ is de *initiële vertakkingsprioriteit*, en $\preceq$ is een *totale pre-orde* relatie over de elementen van $\mathcal{BP}$.

Voor de operationele semantiek $\omega_p^\vee$ breiden we ω_p toestanden uit met een vertakkingsprioriteit. De combinatie van beide noemen we een *alternatief* en wordt genoteerd als $bp :: \sigma$ waarbij bp de vertakkingsprioriteit is en σ een ω_p uitvoeringstoestand. Een alternatief kan *gemarkeerd* zijn, en wordt dan genoteerd als $bp :: \sigma^\star$. Markeringen worden gebruikt om een volgende oplossing te vinden.

Gebruikelijke Zoekstrategieën Specifiëren

We kunnen eenvoudig verschillende gebruikelijke zoekstrategieën implementeren in $\text{CHR}_\vee^{\text{brp}}$. We beschouwen zowel ongeïnformeerde strategieën zoals diepte-eerst, breedte-eerst en diepte-eerst iteratief verdiepen, als geïnformeerde strategieën zoals beste-eerst, A* en beperkte-discrepantie zoeken. Voor ongeïnformeerde strategieën kunnen we automatisch vertakkingsprioriteiten toekennen aan een programma zonder zulke prioriteiten. Tenslotte kunnen we ook strategieën combineren. We hebben als voorbeeld twee combinaties van diepte-eerst en breedte-eerst uitgewerkt. Bij de eerste combinatie wordt er breedte-eerst gezocht tot op een bepaalde

diepte. Bij de tweede combinatie worden alternatieven gegenereerd door één soort regels in een diepte-eerst volgorde doorzocht, terwijl alternatieven gegenereerd door de overige regels in breedte-eerst volgorde doorzocht worden.

Conflict-Gerichte Backjumping

In (Wolf et al. 2007) gebruiken Wolf et al. *justifications* en *conflictverzamelingen* om een *uitgebreide en verfijnde operationele semantiek* $\omega_r^{\vee\star}$ te definiëren. Deze semantiek ondersteunt conflict-gerichte backjumping (CBJ) (Prosser 1993). Op een gelijkaardige wijze stellen we een uitgebreide versie van de $\omega_p^{\vee}$ semantiek voor om CBJ te ondersteunen in ons raamwerk. Terwijl (Wolf et al. 2007) enkel de combinatie van CBJ met links-naar-rechts diepte-eerst zoeken aanbiedt, ondersteunen wij CBJ in combinatie met eender welke zoekstrategie. Bovendien geven we een criterium dat correctheid garandeert, en bestuderen we de optimale conflictkeuze voor het geval dat er zich meerdere conflicten gelijktijdig voordoen.

5 Complexiteitsanalyse van CHR$^{\text{rp}}$-Programma's

In dit hoofdstuk bestuderen we de overeenkomsten tussen de Logical Algorithms taal (LA) van Ganzinger en McAllester, en Constraint Handling Rules. We presenteren een vertaalschema van LA naar CHR$^{\text{rp}}$, en tonen dat het meta-complexiteitsresultaat voor LA kan toegepast worden op een subset van CHR$^{\text{rp}}$ via vertaling in de andere richting. Geïnspireerd door het hoog-niveau implementatievoorstel voor Logical Algorithms door Ganzinger en McAllester, en gebaseerd op een nieuw scheduling algoritme, stellen we een alternatieve implementatie voor CHR$^{\text{rp}}$ voor die sterke complexiteitsgaranties biedt en resulteert in een nieuwe en nauwkeurige meta-complexiteitsstelling voor CHR$^{\text{rp}}$. We tonen bovendien aan dat de vertaling van Logical Algorithms naar CHR$^{\text{rp}}$, gecombineerd met de nieuwe CHR$^{\text{rp}}$-implementatie, aan de complexiteitsvereisten voldoet om het Logical Algorithms meta-complexiteitsresultaat te doen gelden.

Inleiding

Recentelijk werd aangetoond dat alle algoritmes in CHR geïmplementeerd kunnen worden met behoud van zowel de tijds- als ruimtecomplexiteit (Sneyers et al. 2005). In "Logical Algorithms" (LA) (Ganzinger and McAllester 2002) presenteren Ganzinger en McAllester een bottom-up logische programmeertaal met als doel de afleiding van complexiteitsresultaten voor algoritmes die beschreven zijn via logische inferentieregels, te vergemakkelijken. Dit probleem is verre van triviaal omdat de looptijd niet noodzakelijk proportioneel is tot de derivatielengte (t.t.z., het aantal regeltoepassingen), maar ook de kost omvat voor het matchen

van meerhoofdige regels evenals de kosten die volgen uit de hoog-niveau uitvoeringscontrole, die in de Logical Algorithms taal via regelprioriteiten gespecifieerd is. De taal van Ganzinger en McAllester gelijkt op vele vlakken op CHR. Er werd in het verleden dan ook veelvuldig naar verwezen in de discussie van complexiteitsresultaten voor CHR-programma's (Christiansen 2005; Frühwirth 2002b; Schrijvers and Frühwirth 2006; Sneyers et al. 2006a).

Het doel van dit hoofdstuk is om de relatie tussen beide talen te bestuderen. Meer in het bijzonder bekijken we hoe de meta-complexiteitsstelling voor Logical Algorithms kan toegepast worden op (een subset van) CHR, en hoe CHR gebruikt kan worden om Logical Algorithms te implementeren met de juiste complexiteit. Eerst presenteren we een vertalingsschema van Logical Algorithms naar CHR^{rp}. Logical Algorithms derivaties voor het originele programma komen overeen met CHR^{rp}-derivaties in de vertaling en vice versa. We tonen ook hoe een subklasse van CHR^{rp}-programma's in Logical Algorithms kan vertaald worden. Dit laat toe om de meta-complexiteitsstelling voor Logical Algorithms ook toe te passen op deze CHR^{rp}-programma's. Omdat de Logical Algorithms meta-complexiteitsstelling gebaseerd is op een geoptimaliseerde implementatie, geeft deze meer nauwkeurige resultaten dan de implementatie-onafhankelijke stelling van (Frühwirth 2002a; Frühwirth 2002b), en is tegelijkertijd meer algemeen toepasbaar dan de ad-hoc complexiteitsanalyses in (Schrijvers and Frühwirth 2006; Sneyers et al. 2006a).

Onze huidige implementatie van CHR^{rp} zoals gepresenteerd in Hoofdstuk 3, kan de complexiteitsvereisten niet garanderen die nodig zijn om de meta-complexiteitsstelling voor Logical Algorithms te laten gelden via de vertaling naar CHR^{rp}. Een ander probleem is dat de vertaling van CHR^{rp} naar Logical Algorithms beperkt is tot een subset van CHR^{rp}. Daarom stellen we een nieuwe implementatie voor CHR^{rp} voor, die zo ontworpen is dat het een nieuwe meta-complexiteitsstelling voor CHR^{rp} volledig ondersteunt, en tegelijkertijd verzekert dat LA-programma's vertaald naar CHR^{rp} met de juiste complexiteit worden uitgevoerd. We merken op dat deze alternatieve implementatie niet geoptimaliseerd is op performantie in het gemiddelde geval, maar ontworpen is om bepaalde complexiteitsgaranties te kunnen aanbieden.

Meer specifiek is de implementatie gebaseerd op het hoog-niveau implementatievoorstel voor Logical Algorithms zoals gegeven in (Ganzinger and McAllester 2002), en een nieuwe scheduling gegevensstructuur die in detail beschreven is in (De Koninck 2007). De implementatie is beschreven als een vertaling naar gewone CHR. Door een CHR-systeem te gebruiken met geavanceerde indexering, zoals het K.U.Leuven CHR-systeem (Schrijvers and Demoen 2004), voldoet onze implementatie aan de complexiteitsvereisten die nodig zijn om een nieuw en nauwkeurig meta-complexiteitsresultaat voor CHR^{rp} mogelijk te maken.

Logical Algorithms en CHR^{rp}

In deze sectie geven we een overzicht van de syntax en semantiek van Logical Algorithms en bekijken we bestaande meta-complexiteitsresultaten voor zowel LA als CHR.

Een Logical Algorithms programma $P = \{r_1, \ldots, r_n\}$ is een verzameling regels. Een Logical Algorithms regel is een uitdrukking

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

waarbij r de *naam* van de regel is, de atomen A_i (voor $1 \leq i \leq n$) zijn de *antecedenten* en C is het besluit: een conjunctie van atomen waarvan de variabelen in de antecedenten voorkomen. Regel r heeft *prioriteit* p waarbij p een aritmetische uitdrukking is waarvan de variabelen voorkomen in de eerste antecedent A_1. Er zijn twee types van atomen: (aritmetische) vergelijkingen en gebruikersgedefinieerde atomen. Een gebruikersgedefinieerd atoom kan bovendien positief of negatief zijn. Een negatief atoom wordt voorgesteld als `del`(A) met A een positief atoom. Een *ground* gebruikersgedefinieerd atoom wordt een *bewering* (assertion) genoemd.

Een Logical Algorithms uitvoeringstoestand is een verzameling van (positieve en negatieve) beweringen. Een toestand kan gelijktijdig de positieve bewering A en de negatieve bewering `del`(A) bevatten. Een regel kan slechts vuren als zijn conclusie niet reeds in de toestand vervat zit. Een toestand wordt *finaal* genoemd als er geen regels meer op van toepassing zijn.

Voor de Logical Algorithms taal bestaat er een meta-complexiteitsresultaat dat de tijdscomplexiteit weergeeft in functie van het aantal beweringen, *sterke prefixvuringen* en verschillende prioriteiten. Een sterke prefixvuring is hier een prefix van een regelinstantie waarvoor alle antecedenten gelden in een toestand waarin de eerstvolgende regelvuring een prioriteit heeft die maximaal zo hoog is als die van de eerder vermelde regelinstantie.

Voor CHR heeft Frühwirth in (Frühwirth 2002a; Frühwirth 2002b) een meta-complexiteitsresultaat ontwikkeld dat de tijdscomplexiteit weergeeft in functie van het aantal regelvuringen, het maximaal aantal constraints in de CHR constraint store, en kostuitdrukkingen voor het matchen en toevoegen van constraints. Het resultaat is gebaseerd op een zeer naïeve implementatie waarin na elke regelvuring alle regels opnieuw doorlopen worden. We hebben aangetoond dat in afwezigheid van ingebouwde constraints, het resultaat voor LA minstens zo nauwkeurig is als Frühwirth's resultaat voor CHR.

Vertalingen Van en Naar Logical Algorithms en CHR^{rp}

We hebben vertaalschema's ontworpen voor enerzijds de vertaling van Logical Algorithms naar CHR^{rp} en anderzijds de vertaling van een subset van CHR^{rp} naar Logical Algorithms. In de vertaling worden de semantische verschillen tussen beide talen overbrugd. Dit betreft in het bijzonder het verschil tussen set-

en multi-setsemantiek, en het verschil tussen de monotone verwijdering van beweringen in LA en de niet-monotone verwijdering van constraints in CHR. Een volledige vertaling van CHR$^{\text{rp}}$ naar Logical Algorithms is niet mogelijk zonder de ingebouwde constraint theorie te repliceren. Daarom dat we ons beperken tot CHR$^{\text{rp}}$-programma's die geen gebruik maken van ingebouwde constraints.

De vertaling van Logical Algorithms naar CHR$^{\text{rp}}$ laat toe om LA-programma's uit te voeren. Onze CHR$^{\text{rp}}$-implementatie (Hoofdstuk 3) biedt echter niet de complexiteitsgaranties om aan het LA meta-complexiteitsresultaat te voldoen. De vertaling van een subset van CHR$^{\text{rp}}$ naar LA laat toe om het LA meta-complexiteitsresultaat toe te passen op deze vertaalde programma's, en zo een bovengrens op de optimale complexiteit te vinden.

Een Nieuw Meta-Complexiteitsresultaat voor CHR$^{\text{rp}}$

Om bovenvermelde beperkingen op te lossen hebben we een nieuwe implementatie voor CHR$^{\text{rp}}$ ontwikkeld, gebaseerd op het implementatievoorstel voor LA uit (Ganzinger and McAllester 2002) en het scheduling algoritme uit (De Koninck 2007). Deze implementatie leidt tot een nieuw meta-complexiteitsresultaat voor CHR$^{\text{rp}}$. Dit nieuwe resultaat ondersteunt ook ingebouwde constraints. Met behulp van dit resultaat hebben we voor twee niet-triviale programma's een scherpe complexiteitsgrens afgeleid. De vertaling van Logical Algorithms naar CHR$^{\text{rp}}$, uitgevoerd met de nieuwe implementatie, heeft wel de juiste complexiteit. Tenslotte is ons nieuwe resultaat, uitgezonderd enkele speciale gevallen, minstens zo nauwkeurig als het resultaat in (Frühwirth 2002a; Frühwirth 2002b). De uitzondering betreft programma's waarvoor een naïeve aanpak voor het herbekijken van regels na een ingebouwde constraint efficiënter is dan de gespecialiseerde aanpak die wij gebruiken. Voor zulke programma's onderschat Frühwirth's resultaat echter ook de complexiteit in de bestaande geoptimaliseerde implementaties van CHR.

6 Optimalisatie van de Join Order

Join order optimalisatie is het probleem van kostoptimale uitvoeringsplannen te vinden om meerhoofdige regels te matchen. In de context van Constraint Handling Rules heeft dit onderwerp nog maar beperkte aandacht gekregen, ondanks zijn grote belang voor de efficiëntie van de CHR-uitvoering. We presenteren een formeel kostenmodel voor joins en onderzoeken de mogelijkheid van join order optimalisatie tijdens de uitvoering. We stellen enkele heuristische benaderingen voor de parameters van dit kostenmodel voor, zowel voor het statische, als voor het dynamische geval. We bespreken een $\mathcal{O}(n \log n)$ optimalisatie-algoritme voor het speciale geval van acyclische join grafen. In het algemeen is join order optimalisatie echter een NP-volledig probleem. Tenslotte identificeren we enkele klassen van cyclische join grafen die gereduceerd kunnen worden tot acyclische.

Inleiding

Veel werk is reeds besteed aan de geoptimaliseerde compilatie van CHR (Duck 2005; Holzbaur et al. 2005; Schrijvers 2005). Een cruciaal aspect van CHR-compilatie is het efficiënt vinden van toepasbare regels. Gegeven een *actieve* constraint[2], dan komt de zoektocht naar partner constraints overeen met het joinen van relaties — een onderwerp dat in de gegevensbankencontext reeds grondig onderzocht is (Bayardo and Miranker 1996; Krishnamurthy et al. 1986; Selinger et al. 1979; Steinbrunn et al. 1997; Swami and Iyer 1993). De performantie van join methodes hangt af van indexering en join order optimalisatie.

In de context van CHR is join order optimalisatie reeds behandeld in (Duck 2005; Holzbaur et al. 2005). In dat werk wordt enkel statische informatie (gekend tijdens de compilatie) gebruikt om de optimale join order te bepalen. Bovendien is er slechts weinig aandacht geschonken aan de complexiteit van het optimalisatie-algoritme, gezien de optimalisatie gedaan wordt tijdens de compilatie, en omdat de invoergroottes (het aantal hoofden in een regel) worden verondersteld klein te zijn. In dit hoofdstuk beschouwen we ook dynamische join order optimalisatie (t.t.z., tijdens de uitvoering), gebaseerd op statistieken i.v.m. de constraint store zoals selectiviteiten en kardinaliteiten. We houden ook rekening met het feit dat CHR-programma's steeds vaker regels bevatten met een groter aantal hoofden.

De belangrijkste bijdragen van dit hoofdstuk zijn de volgende. We formuleren een algemeen kostenmodel voor het vinden van regelinstanties gegeven een join order, en creëren zo een solide basis voor de ontwikkeling en evaluatie van heuristieken. We introduceren dynamische join order optimalisatie. Tenslotte bespreken we een efficiënt $\mathcal{O}(n \log n)$ join order optimisatie-algoritme voor acyclische join grafen.

Kostenmodel

We introduceren een kostenmodel om de join kost voor een gegeven join order te benaderen. Ons model veronderstelt dat alle regels propagatieregels zijn met lichamen die de relevante statistieken niet fundamenteel veranderen. Dit is gelijkaardig aan wat men in de gegevensbankenliteratuur veronderstelt. In het geval van simplificatie- of simpagatieregels lijkt het voordelig om te optimaliseren voor het vinden van de eerste volledige regelinstanties. In (Bayardo and Miranker 1996) wordt dit onderwerp behandeld door de kans dat een gegeven partiële regelinstantie niet uitgebreid kan worden tot een volledige regelinstantie, in rekening te brengen.

Een join gaat nu als volgt in zijn werk. Een actieve constraint vormt een (mogelijk partiële) regelinstantie. We breiden een partiële regelinstantie uit door

[2]De meeste moderne CHR-implementaties zoeken enkel regelinstanties waarin een gegeven actieve constraint participeert, en maken nieuwe constraints actief om zo alle regelinstanties te vinden.

constraints te zoeken die matchen met het volgende hoofd, gegeven de partiële regelinstantie. Hierbij is de keuze van het volgende hoofd bepaald door de join order. Door indexering kunnen we alle constraints van een bepaald type (functor en ariteit) en gegeven waarden voor een deel van de argumenten, in constante tijd opvragen. We noemen dit *a priori* selectie. Indien na de a priori selectie er nog steeds constraints zijn die niet aan alle wachters voldoen, gebeurt er bovendien nog een *a posteriori* selectie.

We kunnen nu de grote van een partiële regelinstantie bestaande uit k hoofden, neerschrijven als

$$\left|\mathcal{J}_\Theta^k\right| = \left|\mathcal{J}_\Theta^{k-1}\right| \cdot \sigma_{\mathsf{eq}}(k) \cdot \mu(k) \cdot \sigma_\star(k)$$

Hierbij is $\sigma_{\mathsf{eq}}(k)$ de a priori selectiviteit. Een a priori selectie levert gemiddeld $\mu(k)$ constraints op; $\sigma_\star(k)$ is tenslotte de a posteriori selectiviteit. De totale join kost is gelijk aan de som van de groottes van de partiële regelinstanties vóór de a posteriori selectie:

$$C_\Theta^n = \sum_{j=1}^{n} \frac{|\mathcal{J}_\Theta^j|}{\sigma_\star(j)}$$

Benadering van de Parameters

We stellen zowel statische als dynamische benaderingen voor de parameters van ons kost model voor. Vooreerst merken we op dat de selectiviteiten steeds tussen 0 en 1 moeten liggen, en de multipliciteit $\mu(k)$ minstens 1 en maximaal het aantal constraints van het gegeven type is. Indien er een *functionele afhankelijkheid* is tussen de reeds gekende argumenten van de op te zoeken constraint, en de overige argumenten, dan is $\mu(k) = 1$. Een statische analyse om functionele afhankelijkheden te detecteren is gegeven in (Duck and Schrijvers 2005). Voor dynamische parameterbenadering maken we gebruik van statistieken die eenvoudig in de verschillende constraintindexen kunnen worden bijgehouden. We beschouwen zowel absolute grenzen als gemiddelden. Een goede benadering houdt in beperkte mate rekening met de mogelijk sterke afwijkingen van het gemiddelde geval.

Een Optimale Join Order Vinden

In deze sectie bekijken we algoritmes die gebruikt kunnen worden om de optimale join order te vinden, gegeven ons kostenmodel en benaderingen voor zijn parameters. De join grafe voor een gegeven regel bestaat uit knopen die de verschillende hoofden voorstellen, en bogen tussen die hoofden die gelinkt zijn via gemeenschappelijke variabelen. Voor het geval van een acyclische join grafe bestaat er een $\mathcal{O}(n \log n)$ algoritme om de optimale join order te vinden. In het algemeen is het optimalisatieprobleem echter NP-volledig (Ibaraki and Kameda 1984).

7 Algemeen Besluit

In de inleiding hebben we de doelstelling van dit proefschrift beschreven als het ontwerp, de analyse, en de implementatie van taalconcepten en -constructies om zo flexibele, hoog-niveau, en efficiënte uitvoeringscontrole in CHR te ondersteunen. In dit laatste hoofdstuk vatten we onze bijdragen samen en geven we suggesties voor toekomstig onderzoek.

Bijdragen

We vatten onze bijdragen als volgt samen.

- In Hoofdstuk 3 introduceerden we CHR^{rp}: CHR uitgebreid met gebruikersgedefinieerde regelprioriteiten. We hebben met voorbeelden aangetoond hoe CHR^{rp} gebruikt kan worden om geavanceerde propagatiestrategieën op een beknopte manier uit te drukken. We hebben besproken hoe confluentieresultaten voor CHR over te zetten zijn op CHR^{rp} en hebben confluentie onder de ω_p semantiek van CHR^{rp} gelinkt met het concept van observeerbare confluentie. Een basiscompilatieschema voor CHR^{rp} is gepresenteerd en optimalisaties op dit schema zijn voorgesteld. De resulterende CHR^{rp}-implementatie is empirisch geëvalueerd en vergeleken met het state-of-the-art K.U.Leuven CHR-systeem.

- In Hoofdstuk 4 stellen we een nieuw raamwerk voor uitvoeringscontrole in CHR met disjunctie ($\text{CHR}^{\vee}$) voor. Dit raamwerk combineert $\text{CHR}^{\vee}$ en CHR^{rp} en breidt deze uit met prioritaire zoekboomalternatieven (vertakkingsprioriteiten). Het resultaat is een uniforme oplossing voor uitvoeringscontrole in CHR, die de controle van zowel de propagatiestrategie als de zoekstrategie toelaat. We hebben gedemonstreerd hoe verscheidene gebruikelijke zoekstrategieën eenvoudig uitgedrukt kunnen worden in $\text{CHR}_{\vee}^{\text{brp}}$. Dit omvat zowel ongeïnformeerde als geïnformeerde strategieën, evenals combinaties van verschillende strategieën. Tenslotte presenteren we een uitgebreide operationele semantiek voor $\text{CHR}_{\vee}^{\text{brp}}$ om conflict-gerichte backjumping te ondersteunen.

- In Hoofdstuk 5 onderzoeken we de overeenkomst tussen Ganzinger en McAllester's Logical Algorithms formalisme (LA) en CHR^{rp}. LA is een theoretische bottom-up logische programmeertaal, vergezeld van een meta-complexiteitsresultaat dat toelaat de tijdscomplexiteit van LA-programma's te bepalen in termen van het aantal (partiële) regelvuringen en beweringen. Een vertaling van LA naar CHR^{rp} laat toe LA-programma's uit te voeren gebruikmakend van onze CHR^{rp}-implementatie. Een vertaling van een subset van CHR^{rp} naar LA laat toe om het meta-complexiteitsresultaat voor LA toe

te passen op deze CHR$^{\text{rp}}$-programma's. Tenslotte stellen we een alternatieve implementatie voor CHR$^{\text{rp}}$ voor die leidt tot een nieuw en sterk meta-complexiteitsresultaat voor CHR$^{\text{rp}}$ dat algemener is dan het resultaat voor LA en, op enkele uitzonderingen na, scherpere grenzen geeft dan eerdere resultaten voor CHR.

- In Hoofdstuk 6 analyseerden we het join ordering probleem in CHR$^{(\text{rp})}$. Dit probleem bestaat erin een optimale volgorde te vinden om constraints op te zoeken tijdens het matchen met een meerhoofdige regel. We hebben een nieuw, realistischer, model voor de kost van het matchen voorgesteld. We beschouwen ook join order optimalisatie tijdens de uitvoering, hetgeen toelaat om meer nauwkeurige benaderingen voor de parameters van het kostenmodel te maken. Tenslotte hebben we optimalisatie-algoritmes uit de gegevensbankenwereld overgezet naar de CHR context. In het bijzonder is aangetoond dat voor regels met een acyclische join grafe, we een optimale join order kunnen vinden in $\mathcal{O}(n \log n)$ tijd met n het aantal hoofden van de regel. In het algemeen echter is het join order optimalisatieprobleem NP-volledig.

Toekomstig Werk

Het CHR$^{\text{brp}}_{\vee}$-raamwerk is nog niet geïmplementeerd. Een belangrijk aspect van zulk een implementatie is de mogelijkheid om snel tussen verschillende knopen van de zoekboom te springen. Een eenvoudig trailing-mechanisme volstaat hiervoor wellicht niet. De keuze van wat voor soort prioriteitsrij we het beste gebruiken om de alternatieven te ordenen, is afhankelijk van de zoekstrategie die geïmplementeerd wordt. Ook de alternatieve implementatie voor CHR$^{\text{rp}}$ dient verder uitgewerkt te worden. De scheduling gegevensstructuur zoals beschreven in (De Koninck 2007) is reeds geïmplementeerd in Java. Het overige deel zou redelijk eenvoudig moeten zijn.

In Hoofdstuk 3 bespraken we confluentie onder CHR$^{\text{rp}}$'s ω_p semantiek. Deze discussie heeft enkele interessante inzichten opgeleverd, maar een praktische confluentietest ontbreekt nog. In Hoofdstuk 5 werd een systematische aanpak voorgesteld om de tijdscomplexiteit van CHR$^{\text{rp}}$-programma's af te leiden. Andere belangrijke programma-eigenschappen die we nog niet in detail bestudeerd hebben zijn onder meer ruimtecomplexiteit en terminatie.

In deze thesis hebben we het probleem van uitvoeringscontrole in CHR aangepakt. Naast dit controle-aspect kan ook de manier waarop de logica van een constraintoplosser wordt uitgedrukt in CHR, verbeterd worden. Eén resultaat in die richting is gegeven in (Van Weert et al. 2008), alwaar CHR uitgebreid wordt met aggregaten. De ACD Term Rewriting taal (ACDTR) (Duck et al. 2006) is een veralgemening van zowel CHR als AC termherschrijven. Een interessant aspect van ACDTR is dat alle constraints niet noodzakelijk deel uitmaken van één (vlakke) conjunctie zoals in CHR.