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

Analyses, Optimizations and Extensions of Constraint Handling Rules

Promotor :
Prof. Dr. B. DEMOEN

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Tom SCHRIJVERS

Juni 2005

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

Analyses, Optimizations and Extensions of Constraint Handling Rules

Jury :
Prof. Dr. ir. G. De Roeck, voorzitter
Prof. Dr. B. Demoen, promotor
Prof. Dr. ir. M. Bruynooghe
Prof. Dr. ir. F. Piessens
Prof. Dr. D. De Schreye
Prof. Dr. T. Frühwirth (Universität Ulm)
Prof. Dr. D. Warren (SUNY at Stony Brook)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Tom SCHRIJVERS

U.D.C. 681.3*I23

Juni 2005

D/2005/7515/57
ISBN 90–5682–623–9

Abstract

Constraint Handling Rules (CHR) is a rule-based language commonly embedded in a host language. It combines elements of Constraint Logic Programming and term rewriting. Several implementations of CHR exist: in Prolog, Haskell, Java and HAL. Typical applications of CHR are in the area of constraint solving, but currently CHR is also used in a wider range of applications, such as type checking, natural language processing and multi-agent systems.

In this work we contribute program analyses, program optimizations and extensions of the CHR language.

For the optimized compilation of CHR we present several new optimizations: code specialization for ground constraints, anti-monotonic delay avoidance, hashtable constraint stores and a new late storage optimization. These and other optimizations have been implemented in a new state-of-the-art CHR system: the K.U.Leuven CHR system.

Abstract interpretation is a general technique for program analysis. We propose a framework of abstract interpretation for the CHR language, in particular for the formulation of analyses that drive program optimization. This frameworks allows for the uniform formulation of program analyses as well as easier improvements and combinations of existing analyses. We also evaluate analyses for theoretical properties, confluence and time complexity, on a practical case study to investigate their relevance.

We contribute two extensions to the expressivity of CHR. The first extension comprises the integration of CHR with tabled execution. Tabled execution avoids many forms of non-termination and is useful for automatic program optimization through the dynamic reuse of previous computations. The second extension automatically provides implication checking functionality to constraint solvers written in CHR. Implication checking is an essential building block for formulating complex constraints in terms of basic constraints and for composing constraint solvers.

Acknowledgements

First and foremost, I would like to thank my supervisor Bart Demoen for taking me on as his Ph.D. student on short notice and for immediately sending me away again ... to broaden my horizons. Bart has given me the opportunity to work with and learn from many researchers in the field. He has supported all my work with his wise counsel, his unmatched Prolog implementation prowess and his trust in me. A Ph.D. student could not wish for a better supervisor.

I would like to express my gratitude towards the members of the jury for reading and evaluating this text: the members of the reading committee Bart Demoen, Maurice Bruynooghe and Frank Piessens for the valuable comments, Danny De Schreye and the external members David S. Warren and Thom Frühwirth for taking an interest in my work, and Guido De Roeck for chairing the jury.

I would like to thank my co-authors without whom many parts of this text and other joint work would not have been possible: Alexander Serebrenik, Bart Demoen, David S. Warren, Gregory J. Duck, Jan Wielemaker, Jon Sneyers, María García de la Banda, Peter J. Stuckey and Thom Frühwirth. In addition, I would like to thank the other people I have talked and discussed with: Christian Holzbaur, Marc Meister, Marc Wallace, Giridhar Pemmasani, Beata Sarna-Starosta, C.R. Ramakrishnan and Luís Fernando Pias de Castro.

Gratefully I acknowledge the Fund for Scientific Research Flanders (F.W.O.-Vlaanderen), which has provided me with financial support during the last three years.

Many thanks go to my friends, office mates and colleagues in the DTAI research group and the entire department for making the past four years such an enjoyable experience.

Last but not least, I would like to thank Annemie Janssens for all the things beyond code listings and semantic formulas.

Tom Schrijvers
June 2005

Voor Annemie.
Voor Laura Van Driel.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

List of Listings

Chapter 1

Introduction

1.1 Constraint Handling Rules

Constraint Handling Rules (CHR) is a rule-based programming language commonly embedded in a host language. It is a powerful yet relatively simple programming language that combines elements of Constraint (Logic) Programming (CLP) and rule-based languages. Originally CHR was intended as a language for implementing user-defined application-tailored constraint solvers, but it is currently used as a general programming language in a wide range of applications.

First generation CLP languages provided a fixed *black box* constraint solver for each constraint domain, e.g. CLP($\Re$) with its solver over real numbers (Jaffar, Michaylov, Stuckey, and Yap 1992). Second generation solvers have taken a *glass box* approach towards offering more flexibility: the solver offers a number of primitive constraints that can be combined into more complex application-specific constraints. An example is the finite domain constraint solver clp(FD) (Diaz and Codognet 1993) with its primitive `in/2` range constraint that can be used to implement more complex constraints through indexical ranges.

The third is the *no box* generation which allows the implementation of new user-defined constraint solvers from scratch and the integration of different constraint solvers into application-tailored ones. A primitive language feature to provide such functionality in Prolog is attributed variables (Holzbaur 1992). A more recent language in this generation is HAL (Demoen, García de la Banda, Harvey, Marriott, and Stuckey 1999), a variant of Mercury (Somogyi, Henderson, and Conway 1996) that inherited its focus on user declarations for verification and optimization.

CHR belongs to this third generation of constraint solver languages. It is a committed-choice language that consist of multi-headed guarded rules that rewrite constraints into simpler ones until a solved form is reached.

Several implementations of CHR exist and most are embedded in Prolog (IC-

Parc ; Meier ; Holzbaur and Frühwirth 2000; Santos Costa, Damas, Reis, and Azevedo 2004; Holzbaur, García de la Banda, Stuckey, and Duck 2005). CHR implementations for Java (Abdennadher, Krämer, Saft, and Schmauss 2001; Wolf 2001) and Haskell (Stuckey and Sulzmann 2005) exist as well. The implementation (Holzbaur and Frühwirth 2000) in SICStus Prolog is generally considered the reference implementation because it is historically the first full-fledged CHR system. It implements an efficient compilation schema in terms of attributed variables. More recently, the formulation of the refined operational semantics of CHR (Duck, Stuckey, García de la Banda, and Holzbaur 2004) has captured the essentials of the reference implementation on a more formal level.

Typical applications of CHR in the domain of constraint solving, scheduling and optimization, are university lecture time tabling (Abdennadher and Marte 2000) and optimal placement of local telecommunication transmitter stations (Frühwirth and Brisset 1998).

Currently, CHR is used in a wide range of application domains well beyond scheduling and optimization problems of traditional constraint solving. Natural language processing is one such domain, with for example logical grammars based on CHR (Christiansen 2002). Type inference, type checking (Coquery 2003) and type system design (Stuckey and Sulzmann 2005) is a another domain CHR is used in. One more example of a CHR application domain is Multi-Agent Systems: reasoning about hypotheses (Alberti, Chesani, Guerri, Gavanelli, Lamma, Mello, Milano, and Torroni 2005), semantics of agent communication languages (Alberti, Ciampolini, Gavanelli, Lamma, Mello, and Torroni 2003) and modeling the environment of an agent (Thielscher 2005) are just a few of the problems solved with CHR.

1.2 Goal of this Thesis

The main goal of this thesis is to study program analysis, optimized compilation and extensions for the Constraint Handling Rules language:

- In the past, little attention has been given to **optimized compilation** of CHR. The reference implementation of CHR comprises an efficient, yet general compilation schema of CHR and only a small number of program-specific optimizations. Only recently (Holzbaur, García de la Banda, Stuckey, and Duck 2005) was interest raised concerning the application of more substantial optimizations during the compilation of CHR programs and a large number of optimizations for CHR have been formulated.

  In this thesis we present several new optimizations for CHR: code specialization for ground constraints, anti-monotonic delay avoidance, hash table constraint stores and a new late storage optimization. In addition, we formally establish the correctness of several program optimizations with respect

to the refined operational semantics. No such proof of correctness has been published before. The correctness of analyses was addressed only informally.

However, the most important contribution with respect to optimized compilation is our proposal for a more systematic approach of program analysis for optimized compilation.

- **Program analysis** aims at deriving useful properties from a program. In the past, several analyses have been formulated to derive theoretical properties, such as confluence, from CHR programs. These properties are useful for studying the behavior and correctness of CHR programs.

  We apply these analyses of theoretical properties to a case study to investigate their relevance in practice.

  An important class of program analyses are those that drive program optimizations. Little has been published about program analysis for this purpose. Mostly, it is discussed informally and rather vaguely, e.g. in (Holzbaur, García de la Banda, Stuckey, and Duck 2005). The reason is that program analysis was considered in an ad hoc fashion and of lesser importance than the optimization that benefits from it. The lack of proper documentation and rigor makes it difficult to see whether the analyses are correct and how they may be improved and composed to form more complex analyses.

  This is remedied by our framework for abstract interpretation of CHR. Abstract interpretation (Cousot and Cousot 1977) is a general technique for program analysis, not specific to any particular programming language. It is formal in nature and relates an analysis to the operational semantics of the programming language. Because of its rigor, abstract interpretation is an excellent systematic approach towards program analysis for CHR. The framework allows the use of general techniques for establishing correctness and for composing program analyses in CHR.

  We formulate our abstract interpretation framework in terms of the refined denotational semantics and illustrate it with two instances: late storage analysis and groundness analysis. The former is an ad hoc analysis for CHR that we now reformulate in terms of our framework. The latter is a classic analysis for Prolog that we lift to CHR.

- Two of our goals concern **extensions** of the expressive power of CHR.

  The first extension aims at the integration of CHR with a more expressive and powerful variant of Prolog: tabling or SLG resolution (Chen and Warren 1996). Tabling automatically avoids many forms of non-termination of Prolog programs and is also useful for automatic performance improvements through dynamic reuse of previous computations. Because the programmer is relieved from these termination and performance issues, programs may be

more declarative, more concise and easier to understand. We show how the integration of tabling and CHR can be implemented in a general way and easily customized for particular applications.

The second extension adds functionality to constraint solvers written in CHR. CHR constraint solvers answer questions about the satisfiability of constraints, i.e. whether a particular constraint or conjunction of constraints has a feasible solution. We show how to automatically extend CHR constraint solvers with the capability to answer questions about implication, i.e. whether a constraint is implied by a conjunction of other constraints. Implication checking is important for the extraction of information from a constraint store with respect to a property of interest. It is also an important building block for constructing complex application-specific constraints from simple constraints and for composing constraint solvers. We use our implication checking for the modular composition of CHR solvers.

In addition to the main goals we also aim at the increased proliferation of the CHR language and at showing its usefulness in practice. We realize the proliferation objective mainly with a new state-of-the-art CHR system that is available in three Prolog systems, that also helps us to reach and to evaluate our main goals. We show the usefulness of CHR in three applications and through establishing that the union-find algorithm can be implemented in CHR with the best known time complexity.

1.3 Organization of the Text

In this chapter we have briefly summarized the motivation behind the CHR language, its application domains and current shortcomings.

Chapter 2 provides a more thorough technical background for this text. The basic technical aspects of Logic and Constraint Logic Programming are introduced because they form the context in which CHR is embedded. Also covered are the syntax and two different operational semantics of the CHR language.

Three show cases of the CHR language are presented in **Chapter 3**. These show cases illustrate the capabilities and expressive power of the language in three distinct application domains: the union-find algorithm, a framework for Java memory models and the computation of the well-founded semantics.

Chapter 4 provides an overview of three established theoretical properties of CHR programs: the declarative semantics, confluence and time complexity. These three properties are of importance to CHR programmers as they characterize the behavior of CHR programs and may reveal potential undesirable aspects. We illustrate the use of these properties on a case study and point out their limitations and usability issues.

A basic compilation schema to transform a CHR program into executable Prolog code is presented in **Chapter 5**. The schema is related to the refined operational semantics of CHR and extended with the optimizations of the CHR reference implementation. Along with the schema we present two new proofs that establish the correctness of an optimization.

Chapter 6 introduces our contribution to the domain of CHR implementation: the K.U.Leuven CHR system. It covers an overview of the system, the implemented optimizations — both established and novel ones — and the ports to two Prolog systems. The K.U.Leuven CHR system serves as the foundation and test bed for the work in the next chapters.

In **Chapter 7** we formulate an abstract interpretation framework for CHR. This framework provides a systematic approach towards the analysis of CHR programs. We illustrate the use of the framework by instantiating it with two analyses that are useful for the optimized compilation of CHR: late storage analysis and groundness analysis.

The expressive power of CHR is integrated with that of tabled resolution in **Chapter 8**. Our integration is related to the semantics of tabled CLP resolution and various ways to control the basic integration are presented.

Chapter 9 shows how to automatically extend CHR constraint solvers from answering questions about satisfiability to questions about implication. Automatic implication checking is an important building block for composing constraint solvers. We apply our implication checking to CHR constraint solver hierarchies.

The conclusions of this text are listed in **Chapter 10**. It summarizes the contributions contained in the preceding chapters and presents and overview of proposed future work that may be built on top of the presented work.

1.4 Bibliographical Note

Some parts of this work have been published before. The following list contains the key articles. A complete list of publication of the author can be found at the end of this text (page XIX).

The work on a CHR implementation of the union-find algorithm with optimal time complexity, covered in Chapter 4 and Chapter 6, has been accepted by the Theory and Practice of Logic Programming journal:

- T. Schrijvers, and T. Frühwirth, *Optimal Union-Find in Constraint Handling Rules*, Theory and Practice of Logic Programming, 2005.

Our K.U.Leuven CHR systemsystem, described in Chapter 6, has been presented at the First Workshop on Constraint Handling Rules:

- T. Schrijvers, and B. Demoen, *The K.U.Leuven CHR System: Implementation and Application*, First Workshop on Constraint Handling Rules: Selected Contributions (Frühwirth, T. and Meister, M., eds.), pp. 1-5, 2004.

Our framework for abstract interpretation of CHR, discussed in Chapter 7, has been accepted at the 7th International Symposium on Principles and Practice of Declarative Programming:

- T. Schrijvers, G. Duck, and P. Stuckey, *Abstract Interpretation for Constraint Handling Rules*, Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming, Lisbon, Portugal, 2005.

The integration of CHR and tabled execution, see Chapter 8, was presented at the 20th International Conference on Logic Programming and given the Best Paper Award:

- T. Schrijvers, and D. Warren, *Constraint Handling Rules and Tabled Execution*, Logic Programming, 20th International Conference, ICLP 2004, Proceedings (Demoen, B. and Lifschitz, V., eds.), vol 3132, LNCS, pp. 120-136, 2004.

The work on automatic implication checking for CHR constraint solvers, covered in Chapter 9, was presented at the 6th International Workshop on Rule-Based Programming:

- T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth, *Automatic Implication Checking for CHR Constraints,* Proceedings of 6th International Workshop on Rule-Based Programming, Nara, Japan (Cirstea, H. and Marti-Oliet, N., eds.), 2005.

In addition to the work on Constraint Handling Rules, presented in this text, we have worked on two unrelated topics during the course of the doctorate.

The first topic is the trailing of logical variables using the PARMA representation. Two contributions have been made. Firstly, we have developed a program analysis to find redundant trailing operations and the corresponding optimization to get rid of them; both were implemented in HAL. Secondly, more space-efficient trailing operations have been proposed. On average, these two improvements together bring the time and space costs of the PARMA variable representation on the same level as the classical WAM representation.

The main publication that covers this work is:

- T. Schrijvers, M. García de la Banda, B. Demoen and P. Stuckey. *Improving PARMA Trailing.* Theory and Practice of Logic Programming.

The second topic concerns refactoring of Prolog programs. Refactoring is a technique that originates in Object-Oriented Programming. It comprises techniques to improve qualitative aspects of software, such as readability, maintainability and extendibility, without changing functionality. In our work we have adapted the ideas of refactoring to Prolog: object oriented refactoring techniques have been modified to suit Prolog and new Prolog-specific techniques have been proposed. The principal publication that addresses this work is:

- T. Schrijvers and A. Serebrenik. *Improving Prolog programs: refactoring for Prolog,* Logic Programming, 20th International Conference, ICLP 2004, Proceedings (Demoen, B. and Lifschitz, V., eds.), vol 3132, LNCS, pp. 120-136, 2004.

Chapter 2

Background: Constraint Logic Programming and CHR

2.1 Introduction

This chapter contains the necessary background, both notation and semantics, for understanding the core of this thesis. First, in Section 2.2, we present some notational aspects regarding sequences and multisets. Next, in Section 2.3 we cover several aspects of Constraint Logic Programming and in Section 2.4 the syntax and semantics of CHR are covered. Finally, Section 2.5 situates CHR in the context of other rule-based languages.

2.2 Notation of Sequences and Multisets

We use $[H|T]$ to denote a sequence with first element H and remaining elements T, $+\!\!+$ for sequence concatenation and $\square$ for the empty sequence. We use $\bar{s} = \bar{t}$, where $\bar{s}$ and $\bar{t}$ are sequences, to denote the conjunction $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$.

The symbol $\uplus$ is used for multiset union. We shall sometimes treat multisets as sequences, in which case we non-deterministically choose an order for the objects in the multiset.

2.3 Constraint Logic Programming

2.3.1 Logic Programming and Prolog

We give a brief overview of the basic elements of Logic Programming (LP). For a more elaborate introduction to logic programming we refer to (Lloyd 1987).

Syntactic Elements

The basic alphabet of a logic program consists of a set of variables $\mathcal{V}$, a set of function symbols Σ and a set of predicate symbols Π. Function and predicate symbols are associated with an *arity*, a natural number identifying the number of *arguments* the function or predicate symbol has. Function symbols with arity zero are called *constants*. Function and predicate symbols are also called *functors*.

The following syntax is used:

- Variables are denoted by upper case letters or capitalized words, e.g.

  `X`, `Y`, `A1`, `A2`, `Var`, `NewList`

- Function and predicate symbols are denoted by lower case letters and uncapitalized words, e.g.

  `x`, `y`, `a1`, `a2`, `var`, `newList`

When the arity of the function or predicate symbol matters, we explicitly write f/n to denote the function or predicate symbol f with arity n.

A *term* is a variable or a compound term $f(\tau_1, \ldots, \tau_n)$ where $f/n \in \Sigma$ and each argument τ_i is a term. The set of all terms is $\mathcal{T}(\mathcal{V}, \Sigma)$. An *atom* is a predicate symbol $p/n \in \Pi$ applied to a sequence of terms. The predicate symbol `=/2` is a special predicate symbol called *(explicit) unification*.

A *ground term* is a term that does not contain any variables. A *ground atom* is an atom that does not contain any variables.

We use the notion of *expression* to refer to both terms and atoms. The relation symbol $\equiv$ denotes that two expressions E and F are syntactically equal, to avoid confusion with explicit unification. The function $vars(E)$ returns the set of variables occurring in an expression E.

The notation $\bar{\exists}_A F$ is used to denote the projection of logic formula F onto the variables of syntactic object A. Formally,

$$\bar{\exists}_A F \equiv \exists X_1 \cdots \exists X_n F \text{ with } \{X_1, \ldots X_n\} = vars(F) - vars(A)$$

Variable Substitutions

A *substitution* θ is a finite mapping from distinct variables to terms: $\mathcal{V} \mapsto \mathcal{T}(\mathcal{V}, \Sigma)$. A substitution is denoted as:

$$\theta = \{X_1/\tau_1, \ldots, X_n/\tau_n\}$$

where each $X_i \neq \tau_i$. Each element X_i/τ_i is called a *binding*. If E is an expression and θ a substitution, then $E\theta$ is the expression obtained by simultaneously replacing each occurrence of a variable X_i in E by the term τ_i, for each $X_i/\tau_i \in \theta$. $E\theta$ is called an *instance* of E.

If $\theta = \{X_1/u_1, \ldots, X_n/u_n\}$ and $\eta = \{Y_1/v_1, \ldots, Y_m/v_m\}$ are two substitutions, then the *composition* of η and θ (denoted $\eta * \theta$) is a substitution obtained from $\{Y_1/v_1\theta, \ldots, Y_m/v_n\theta, X_1/u_1, \ldots, X_n/u_n\}$ by removing all elements $Y_i/v_i\theta$ such that $v_i\theta = Y_i$ and all elements X_i/u_i such that X_i is one of $Y_1, \ldots, Y_m$.

If E and F are expressions, then E and F are *variants* of each other if there exist two substitutions θ and σ such that $E \equiv F\sigma$ and $E\theta \equiv F$. The notation $E \sim F$ is used to denote that E and F are variants. A *renamed apart* expression F of an expression E is a variant of E for which $vars(F) \cap vars(E) = \emptyset$. We usually abbreviate the phrase "a renamed apart expression F of expression E" to "a renamed apart expression E".

A *unifier* of two expressions E and F is a substitution θ for which $E\theta \equiv F\theta$. The *most general unifier* (mgu) of two expressions E and F is a unifier η such that for any other unifier θ of E and F there exists a substitution ϕ such that $\theta = \eta * \phi$.

Logic Programs

A *logic program* consists of a number of rules, called *clauses*, of the form:

$$H \texttt{:-} L_1 \texttt{,} \ldots \texttt{,} L_n \texttt{.}$$

where $n \geq 0$ and H is an atom and $L_1, \ldots, L_n$ are *literals*. A literal is either an atom A or a negated atom $\neg A$.

H is called the *head* of the clause and $L_1, \ldots, L_n$ is called the *body*. The comma "`,`" is called *conjunction* as it corresponds with logical conjunction in the semantics of logic programs. If $n = 0$, then the clause is also called a *fact* and denoted as:

$$H \texttt{.}$$

If all the literals in the body are positive, the clause is a *definite clause*. A *normal clause* is a clause that may also contain negative literals. A *definite logic program* consists of definite clauses only, while a *normal logic program* has normal clauses.

Various semantics for logic programs exist. These assign a truth value to all literals that can be constructed from predicate and function symbols in a program. In Section 3.4 we will study one such semantics, the well-founded semantics, in more detail.

Parent	Children	Conditions
Clause Resolution		
$(G; A_1, \ldots, A_n)$	$(G; B_1^1, \ldots, B_k^1, A_2, \ldots, A_n)\theta_1$	for all $0 < i \leq l$ such that
	$\vdots$	$H_i \rightarrow B_1^i, \ldots, B_{k_i}^i \in P$
	$(G; B_1^l, \ldots, B_{k_l}^l, A_2, \ldots, A_n)\theta_l$	and θ_i the mgu of A_1 and H_i
Explicit Unification		
$(G; X = Y, A_2, \ldots, A_n)$	$(G; A_2, \ldots, A_n)\theta$	θ the mgu of X and Y

Table 2.1: Rewriting rules for SLD resolution with left-to-right selection

Operational Semantics of Logic Programs

A popular operational semantics of logic programs is *SLD resolution*. SLD resolution stands for **L**inear resolution with a **S**election function applied to **D**efinite clauses. Given a conjunction of literals G, called a *goal* or *query*, SLD resolution determines a number of unifiers θ such that $G\theta$ holds. $G\theta$ is called an *answer* to the goal G.

SLD resolution is parametrized in a selection function, which selects a next literal in the resolvent. The most common selection function is *left-to-right selection* that resolves literals in the *resolvent* from left to right.

Table 2.1 lists the rewriting rules for SLD resolution with left-to-right selection and explicit unification. The rules manipulate $(G; R)$ where G is the initial goal and R is the resolvent and define a tree with root $(G; G)$. Answers to the initial goal are contained in leaves of the form $(G'; \Box)$. In such a leaf $G' = G\theta$ is an instance of G derived from G through successive application of unifiers. The path from the root to a leaf is a single SLD resolution. If the SLD-tree for a goal has at least one answer, we say the goal *succeeds*. If the tree has no answers, we say the goal *fails*. A goal whose tree has at most one answer, is *deterministic*, otherwise it is *non-deterministic*.

A concrete implementation of SLD resolution needs to supply a particular tree search strategy to explore the SLD-tree.

SLDNF resolution (SLD with negation as failure) is an extension of SLD resolution to normal logic programs. For the atom of a negative literal in a resolvent a separate SLD resolution tree is derived. If this separate tree fails to produce any answers, the negative literal is dropped from the resolvent.

Prolog

Prolog implements SLDNF resolution for normal logic programs with left-to-right selection and depth-first search. A node in the SLD-tree with multiple children is called a *choice-point* in Prolog terminology and returning to a higher-up node after exploring a subtree is called *backtracking*. During backtracking Prolog undoes the effect of unifiers applied in the explored subtree.

Besides the terms manipulated in logic programs, Prolog also adds numeric data: integers and floating point numbers. For most purposes, these behave in the same way as constants.

In addition to user-defined predicates and explicit unification Prolog provides a number of predefined predicates, called *built-in predicates* or simply *built-ins*. These built-ins are described in the ISO specification of Prolog (ISO/IEC 1995). The following are a few of the built-ins used in this text:

`true/0` Always succeeds.

`fail/0` Never succeeds.

`var/1` Succeeds if its argument is a variable.

`nonvar/1` Succeeds if its argument is not a variable.

`ground/1` Succeeds if its argument is ground.

`term_variables/2` Succeeds and unifies its second argument with a list of variables occurring in the first argument. The second argument should be a variable prior to resolution.

`==/2` Succeeds if its two arguments are syntactically identical. Its opposite is `\==` which succeeds if its two arguments are syntactically different.

`>=/2` Succeeds if both its arguments are arithmetic formulas and the value that the first argument evaluates to is greater or equal to the value that the second argument evaluates to.

`is/2` Unifies the left-hand side (first argument) with the numeric value of the arithmetic formula represented by the right-hand side (second argument). The right-hand side should be a well-formed ground arithmetic expression.

Prolog defines several operators besides conjunction that influence the shape of the SLDNF-tree:

`(`G_1 `;` G_2`)` *Explicit disjunction* creates an additional choice-point, with resolvents G_1 and G_2 along either branch.

`(`C `->` T `;` E`)` *If-then-else* first resolves the condition C. If it succeeds, T is resolved next. Otherwise E is resolved.

`!` *Cut* prunes all alternative branches that are to the right in the SLD-tree on the path between the resolution of the cut and the resolution of the clause in which it appears (inclusive).

`\+ G` Negation as failure constructs a new SLD-tree with goal G and succeeds if the new tree contains no answers.

`findall(Pattern,Goal,List)`
Findall unifies `List` with a list of elements `Pattern`θ for each θ that is an answer substitution of the goal `Goal`.

2.3.2 Constraint Logic Programming

Constraint Logic Programming (CLP) combines Logic Programming with constraint solving. A good survey is given in (Jaffar and Maher 1994) and a recommended introductory work is (Marriott and Stuckey 1998).

Constraint Domains

A *constraint domain* $\mathcal{D}$ consists of a set $\Pi' \subset \Pi$ of constraint symbols, a logical theory $\mathcal{T}$ and for every constraint symbol $c/n \in \Pi'$ a tuple of value sets $\langle V_1, \ldots, V_n \rangle$. A *primitive constraint* is constructed from a constraint symbol c/n and for every argument position i ($1 \leq i \leq n$) either a variable or a value from the corresponding value set V_i, similar to the way an atom is constructed in a logic program.

A *constraint* is of the form $c_1 \wedge \ldots \wedge c_n$ where $n \geq 0$ and $c_1, \ldots, c_n$ are primitive constraints. Two distinct constraints are *true* and *false*. The former always holds and the latter never holds. The empty conjunction of constraints is written as *true*.

The logical theory $\mathcal{T}$ determines what constraints hold and what constraints do not hold. Typically, we use $\mathcal{D}$ to also refer specifically to $\mathcal{T}$. For example $\mathcal{D} \models c$ means that under the logical theory $\mathcal{T}$ of constraint domain $\mathcal{D}$ the constraint c holds.

A *valuation* θ for a constraint C is a variable substitution that maps the variables in $vars(C)$ onto values of the constraint domain $\mathcal{D}$. If θ is a valuation for C, then it is a *solution* for C if $C\theta$ holds in the constraint domain $\mathcal{D}$, i.e. $\mathcal{D} \models C\theta$. A constraint C is *satisfiable* if it has a *solution*; otherwise it is *unsatisfiable*. Two constraints C_1 and C_2 are *equivalent*, denoted $\mathcal{D} \models C_1 \leftrightarrow C_2$, if and only if they have the same solutions.

Two problems associated with a constraint C are the *solution problem*, i.e. determining a particular solution, and the *satisfaction problem*, i.e. determining

whether there exists at least one solution. An algorithm for determining the satisfiability of a constraint is called a *constraint solver*. Often a solution is produced as a by-product. A general technique used by many constraint solvers is to repeatedly rewrite a constraint into an equivalent constraint until a *solved form* is obtained. A constraint in solved form has the property that it is clear whether it is satisfiable or not.

Constraint Logic Programming

The bodies of clauses of a constraint logic program may contain constraints in addition to the ordinary literals.

The operational semantics of logic programs is extended to interoperate with a constraint solver. Throughout resolution a conjunction of constraints encountered so far, the *constraint store* S, is maintained. Initially this constraint store is empty, i.e. $S = \mathit{true}$. Each time a constraint literal c is selected this constraint is added to the constraint store. Logically this addition replaces the old constraint store S_0 with $S_1 = S_0 \wedge c$. The constraint solver then rewrites the resulting constraint store to its solved form. If this solved form is detected to be unsatisfiable, resolution fails. Otherwise, ordinary resolution resumes.

Besides the ordinary constraints that are added to the constraint store, in this context called *tell constraints*, also ask versions of tell constraints (or *ask constraints* for short) may be used in CLP programs. When an ask constraint is selected, it is not added to the constraint store. Instead it is verified whether the constraint store entails the constraint. If it does, resolution resumes. Otherwise resolution fails.

Unification as the Term Equality Constraint

Here we present a constraint theory that we will use throughout this thesis, namely Prolog's equality theory. Prolog's unification ($=$) is a constraint extension of term equality ($\equiv$): it constrains terms to be syntactically equal. The axioms of unification, or the Herbrand constraint theory, $\mathcal{H}$ are the following:

(reflexivity)	$x = x$
(symmetry)	$x = y \rightarrow y = x$
(transitivity)	$x = y \wedge y = z \rightarrow x = z$
(recursion)	$x = f(x_1, \ldots, x_n) \wedge x = g(y_1, \ldots, y_m) \rightarrow$ $f \equiv g \wedge n = m \wedge x_1 = y_1 \wedge \ldots \wedge x_n = y_n$

The `==/2` predicate is the ask version of the `=/2` constraint.

On top of unification some more predicates and a function may be defined:

(nonvar)	$C \rightarrow nonvar(x) \Leftrightarrow \exists f, n, x_1, \ldots, x_n : C \rightarrow x = f(x_1, \ldots, x_n)$
(var)	$C \rightarrow var(x) \Leftrightarrow C \not\rightarrow nonvar(x)$
(ground)	$C \rightarrow ground(x) \Leftrightarrow \nexists v : C \rightarrow v \in term\_vars(x)$
(term_vars)	$C \rightarrow v \in term\_vars(t) \Leftrightarrow \exists f, t_i :$ $C \rightarrow var(v) \wedge (t = v \vee (t = f(\ldots, t_i, \ldots) \wedge v \in term\_vars(t_i))$

where C is any conjunction of unifications. These predicates correspond with Prolog built-ins of the same name and the $term\_vars(t)$ function corresponds to the `term_variables/2` built-in.

The following theorem establishes the relation between equality constraints and substitutions, both of which are used to make terms equal:

Theorem 2.1

$$\forall \theta = \{X_1/\tau_1, \ldots, X_n/\tau_n\} : A \equiv B\theta \leftrightarrow (X_1 = \tau_1 \wedge \cdots \wedge X_n = \tau_n \rightarrow A = B)$$

where $X_1, \ldots, X_n$ are variables and $A, B, \tau_1, \ldots, \tau_n$ are terms.

In an abuse of syntax we will sometimes use θ as the conjunction of equality constraints $X_1 = \tau_1 \wedge \cdots \wedge X_n = \tau_n$.

2.4 Constraint Handling Rules

2.4.1 Syntax

CHR Constraints

CHR constraint symbols are drawn from the set of predicate symbols, denoted by a functor/arity pair. CHR constraints, also called constraint atoms or constraints for short, are atoms constructed from these symbols.

Built-in Constraints

CHR is embedded in a host language that provides a number of predefined constraints. These constraints are called host language constraints or *built-in* constraints. The typical host language of CHR is Prolog and the built-in constraints are unification, Prolog built-ins and user-defined predicates. We will assume that the built-in solver supports at least term equality. The constraint domain of the built-in constraints is denoted by $\mathcal{D}_b$.

Programs and Rules

A CHR program $\mathcal{P}$ consists of an ordered sequence of CHR rules. The ordering is the one in which the rules are written.

There are three different kinds of rules. A *simplification* rule is of the form:

$$c_1, \ldots, c_n <=> g \mid d_1, \ldots, d_m.$$

A *propagation rule* is of the form:

$$c_1, \ldots, c_n ==> g \mid d_1, \ldots, d_m.$$

A *simpagation rule* is of the form:

$$c_1, \ldots, c_l \setminus c_{l+1}, \ldots, c_n <=> g \mid d_1, \ldots, d_m.$$

Here $l, m, n > 0$. The sequence, or conjunction, $c_1, \ldots, c_n$ are CHR constraint atoms; together they are called the *head* or *head constraints* of the rule. A rule with n head constraints is named an *n-headed* rule and when $n > 1$, it is a *multi-headed* rule.

A rule is optionally preceded by *name* @ where *name* is a term. No two rules may have the same name.

All the head constraints of a simplification rule and the head constraints $c_{l+1}, \ldots, c_n$ of a simpagation rule are called *removed* head constraints. The other head constraints, all those of a propagation rule and $c_1, \ldots, c_l$ of a simpagation rule, are called *kept* head constraints.

With every head constraint a number is associated, called the *occurrence*. Head constraints are numbered per functor/arity pair, starting from 1, from the first rule to the last rule, from left to right. Removed heads in a simpagation rule are numbered before kept heads.

The conjunction $d_1, \ldots, d_n$ are either CHR constraints or host language constraints; together they are called the *body* of the rule.

The part of the rule between the arrow and the body, g, is called the *guard*. It is a conjunction of host language ask constraints. The guard "g |"is optional; if omitted, it is considered to be *true*.

We denote the set of all possible CHR programs by **Prog**.

Example 2.1 The following is an example of a CHR program $\mathcal{P}$.

Listing 2.1 - The gcd Program

```
gcd1 @ gcd(0) <=> true.
gcd2 @ gcd(I) \ gcd(J) <=> J >= I | K is J - I, gcd(K).
```

The rules in $\mathcal{P}$ are named respectively `gcd1` and `gcd2`. They are respectively a simplification rule and a simpagation rule.

Generalized Simpagation Form For simplicity, we sometimes consider both simplification and propagation rules as special cases of a simpagation rules. The general form of a *simpagation* rule is:

$$c_1, \ldots, c_l \setminus c_{l+1}, \ldots, c_n <=> g \mid d_1, \ldots, d_m.$$

or

$$H_k \setminus H_r <=> g \mid B$$

where H_r is the sequence of the removed head constraints and H_k is the sequences of the kept CHR constraints, g is a sequence of built-in constraints, and B is a sequence of constraints. If H_k is empty, then the rule is a *simplification* rule. If H_r is empty, then the rule is a *propagation* rule. At least one of H_r and H_k must be non-empty.

Code Listings

All code listed in this text is valid in SWI-Prolog (Wielemaker 2004). Part of the code is ordinary Prolog code and part of it is CHR code.

CHR code in SWI-Prolog always has to be preceded by the following declaration in a source file:

```
:- use_module(library(chr)).
```

to import the CHR library.

In addition all functor/arity pairs of the CHR constraints have to be declared with a `constraints/1` declaration. In this way it is possible to declare CHR constraints that do not occur in the head of a rule.

Example 2.2 The following code listing combines the CHR rules of Example 2.1 in a single program with the required declarations:

```
:- use_module(library(chr)).

:- constraints gcd/1.

gcd1 @ gcd(0) <=> true.
gcd2 @ gcd(I) \ gcd(J) <=> J >= I | K is J - I, gcd(K).
```

2.4.2 The ω_t Operational Semantics

In this section we present the operational semantics ω_t of CHR, sometimes also called *theoretical* or *high-level* operational semantics. The refined operational semantics is discussed in Section 2.4.3.

Several versions of the operational semantics have already appeared in the literature, e.g. (Abdennadher 1997; Frühwirth 1998), essentially as a multiset rewriting semantics. We adopt the version of (Duck, Stuckey, García de la Banda, and Holzbaur 2004), which is equivalent to the previous ones, but closer in formulation to the refined semantics that we will present and use later in this text.

The ω_t semantics is formulated as a state transition system. Transition rules define the relation between an execution state and its subsequent execution state.

Execution State Firstly, we define an *execution state* σ as the tuple $\langle G, S, B, T\rangle_n$. The first part of tuple, the *goal* G is the multiset of constraints to be rewritten to solved form. The CHR constraint *store* S is the multiset of *identified* CHR constraints that can be matched with rules in the program P. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i, the *constraint identifier*. This number serves to differentiate among copies of the same constraint. We introduce the functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner. We also define the functions on multisets of identified constraints in the obvious manner, e.g. $chr(S) = \{c|c\#i \in S\}$.

The *built-in constraint store* B is the conjunction of all built-in constraints that have been passed to the underlying solver. Since we will usually have no information about the internal representation of B, we will model it as an abstract logical conjunction of constraints. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. This is necessary to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used to fire the same rule before. Finally, the counter n represents the next free integer that can be used to number a CHR constraint.

Given an initial goal G, the *initial execution state* is: $\langle G, \emptyset, true, \emptyset\rangle_1$.

Transition Rules The theoretical operational semantics ω_t is based on the three transition rules listed in Figure 2.1 that map execution states to execution states.

The first rule tells the underlying solver to add a new built-in constraint to the built-in constraint store. The second adds a new identified CHR constraint to the CHR constraint store. The last one chooses a program rule for which matching constraints exist in the CHR constraint store, and whose guard is entailed by the underlying solver, and *fires* it. In examples, we usually apply the resulting matching substitution θ to all relevant fields in the execution state, i.e. G, S and B.

The transitions are non-deterministically applied, starting from the initial execution state, until either no more transitions are applicable (a successful derivation), or the underlying solver can prove $\mathcal{D}_b \models \neg\bar{\exists}_\emptyset B$ (a failed derivation). In

1. Solve $\langle \{c\} \uplus G, S, B, T \rangle_n \rightarrowtail_{solve} \langle G, S, c \wedge B, T \rangle_n$ where c is a built-in constraint.

---

2. Introduce $\langle \{c\} \uplus G, S, B, T \rangle_n \rightarrowtail_{introduce} \langle G, \{c\#n\} \uplus S, B, T \rangle_{(n+1)}$ where c is a CHR constraint.

---

3. Apply $\langle G, H_1 \uplus H_2 \uplus S, B, T \rangle_n \rightarrowtail_{apply} \langle C \uplus G, H_1 \uplus S, \theta \wedge B, T' \rangle_n$ where there exists a (renamed apart) rule in $\mathcal{P}$ of the form

$$r \ @ \ H_1' \setminus H_2' \iff g \mid C$$

and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$ and $\mathcal{D}_b \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$. In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [r]\}$. It should hold that $T' \neq T$.

Figure 2.1: The transition rules of the theoretical operational semantics ω_t

both cases a *final state* has been reached, the former a *success state* and the latter a *failed state*.

Example 2.3 Table 2.2 depicts a (terminating) derivation under ω_t for the query `gcd(6), gcd(9)` executed on the `gcd` program in Example 2.1.

Every row in the table indicates an execution state; for brevity, B and T have been omitted. The column with label "Transition" indicates for each execution state the transition used to derive it from the previous state. In case of an apply transition, the column "Rule" indicates the CHR rule used. In every state the constraints are underlined that are affected by the transition to the next state.

No more transition rules are possible after the last state, so this is the final state.

The transition arrow $\rightarrowtail$ is denoted without a subscript when the actual transition rule used is unspecified. The transitive closure of $\rightarrowtail$ is denoted as $\rightarrowtail^*$.

Non-determinism In general for the same initial state many different final states may be reached through many different *derivations*. We denote a particular derivation d from initial state σ_0 to final state σ as $\sigma_0 \rightarrowtail^d \sigma$. Similarly, we denote the final state obtained through a derivation d from an initial state σ_0 as $solve_d(\sigma_0)$. When the particular derivation is irrelevant, it may be omitted, e.g. as in $solve(\sigma_0)$.

Transition	Rule	G	S	n
		$\{\underline{\texttt{gcd}(6)}, \texttt{gcd}(9)\}$	$\emptyset$	1
introduce		$\{\underline{\texttt{gcd}(9)}\}$	$\{\texttt{gcd}(6)\#1\}$	2
introduce		$\emptyset$	$\{\underline{\texttt{gcd}(6)\#1, \texttt{gcd}(9)\#2}\}$	3
apply	gcd2	$\{\underline{O \texttt{ is } 9-6}, \texttt{gcd}(O)\}$	$\{\texttt{gcd}(6)\#1\}$	3
solve		$\{\underline{\texttt{gcd}(3)}\}$	$\{\texttt{gcd}(6)\#1\}$	3
introduce		$\emptyset$	$\{\underline{\texttt{gcd}(6)\#1, \texttt{gcd}(3)\#3}\}$	4
apply	gcd2	$\{\underline{O \texttt{ is } 6-3}, \texttt{gcd}(O)\}$	$\{\texttt{gcd}(3)\#3\}$	4
solve		$\{\underline{\texttt{gcd}(3)}\}$	$\{\texttt{gcd}(3)\#3\}$	4
introduce		$\emptyset$	$\{\underline{\texttt{gcd}(3)\#3, \texttt{gcd}(3)\#4}\}$	5
apply	gcd2	$\{\underline{O \texttt{ is } 3-3}, \texttt{gcd}(O)\}$	$\{\texttt{gcd}(3)\#3\}$	5
solve		$\{\underline{\texttt{gcd}(0)}\}$	$\{\texttt{gcd}(3)\#3\}$	5
introduce		$\emptyset$	$\{\texttt{gcd}(3)\#3, \underline{\texttt{gcd}(0)\#5}\}$	6
apply	gcd1	$\emptyset$	$\{\texttt{gcd}(3)\#3\}$	6

Table 2.2: Example derivation under the ω_t semantics

Head Normal Form For reasons of simplicity, we will sometimes only consider CHR programs in *Head Normal Form*. A CHR program P is in Head Normal Form if every rule r in P is in Head Normal Form. We say that a CHR rule r is in Head Normal Form if all arguments of constraints in the head are unique variables.

A procedure to transform any CHR program into an equivalent program under ω_t in Head Normal Form is given in (Holzbaur, García de la Banda, Stuckey, and Duck 2005).

Example 2.4 For example, the following program is not in Head Normal Form:

```
r1 @ fibonacci(N,M1) \ fibonacci(N,M2) <=> M1 = M2.
r2 @ fibonacci(0,M) ==> M = 1.
r3 @ fibonacci(1,M) ==> M = 1.
r4 @ fibonacci(N,M) ==> N > 1 |
                N1 is N-1, fibonacci(N1,M1),
                N2 is N-2, fibonacci(N2,M2),
                M is M1 + M2.
```

because rules $r1$, $r2$ and $r3$ are not in Head Normal Form. In rule $r1$ the same variable, N, appears twice in the head and in rules $r2$ and $r3$ two arguments are not variables, but numbers.

Example 2.5 However, the following program is in Head Normal Form and equivalent to the first:

```
r1 @ fibonacci(N1,M1) \ fibonacci(N2,M2) <=> N1 == N2
                                  | M1 = M2.
r2 @ fibonacci(N,M) ==> N == 0 | M = 1.
r3 @ fibonacci(N,M) ==> N == 1 | M = 1.
r4 @ fibonacci(N,M) ==> N > 1  |
                N1 is N-1, fibonacci(N1,M1),
                N2 is N-2, fibonacci(N2,M2),
                M is M1 + M2.
```

2.4.3 The ω_r Operational Semantics

In this section we introduce the refined operational semantics ω_r of CHR, defined in (Duck, Stuckey, García de la Banda, and Holzbaur 2004) and implemented in all major CHR systems.

The *refined* operational semantics establishes an order for the constraints in G. As a result, we are no longer free to pick any constraint from G to either **Solve** or **Introduce** into the store. It also treats CHR constraints as procedure calls: each newly added constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires other CHR constraints might be executed and, when they finish, the execution returns to finding rules for the current active constraint. Not surprisingly, this approach is used exactly because it corresponds closely to that of the stack-based programming languages to which CHR is compiled.

Formally, the execution state of the refined semantics is the tuple $\langle A, S, B, T\rangle_n$ where S, B, T and n, representing the CHR store, built-in store, propagation history and next free identity number respectively, are exactly as before. The *execution stack* A is a sequence of constraints, identified CHR constraints and occurrenced identified CHR constraints, with a strict ordering in which the top-most constraint is called the *active constraint*. An *occurrenced* identified CHR constraint $c\#i : j$ indicates that only matches with occurrence j of constraint c should be considered when the constraint is active. Unlike in the theoretical operational semantics, the same identified constraint may simultaneously appear in both the execution stack A and the store S.

Given initial goal G, the initial state is as before, except that G should be a sequence instead of a multiset. Just as with the theoretical operational semantics, execution proceeds by exhaustively applying transitions to the initial execution

state until the built-in solver state is unsatisfiable or no transitions are applicable. The possible transitions are listed in Figure 2.2.

The ω_r semantics is an instance of the ω_t semantics. In (Duck, Stuckey, García de la Banda, and Holzbaur 2004) a mapping is given which maps execution states of ω_r to execution states of ω_t and ω_r derivations to ω_t derivations.

The refined operational semantics is still *non-deterministic*. The first source of non-determinism is the **Solve** transition where the order in which constraints S_1 are added to the activation stack is left unspecified. The definition of **Solve** (which considers all non-fixed CHR constraints) is weak. In practice, only constraints that may potentially cause a new rule to fire are re-added (see (Duck, Stuckey, García de la Banda, and Holzbaur 2003; Holzbaur and Frühwirth 1999) for more details). We say that the CHR constraints added during a **Solve** transition of a built-in constraint are *triggered* by the built-in constraint.

The other sources of non-determinism are present within the **Simplify** and **Propagate** transitions, where we do not know which *partner* constraints (H_1, H_2 and H_3) may be chosen for the transition, if more than one possibility exists. If a **Simplify** or **Propagate** transition occurs, we say that the corresponding CHR rule is *fired*.

Both sources of non-determinism could be removed by further refining the operational semantics. However, the non-determinism is used to model implementation specific behavior of CHRs. For example, different CHR implementations use different data structures to represent the store, and this may affect the order in which partner constraints are matched against a rule. By leaving matching order unspecified, we capture the semantics of more implementations. It also leaves more freedom for optimization of CHR execution (see e.g. (Holzbaur, García de la Banda, Stuckey, and Duck 2005)).

Example 2.6 Table 2.3 shows the derivation under ω_r semantics for the `gcd` program in Example 2.1 and the goal `gcd(6),gcd(9)`. For brevity B and T have been eliminated and the matching substitutions θ applied throughout.

2.5 CHR versus Other Rule-based Languages

Although CHR was conceived in the context of Constraint Logic Programming, it has many characteristics in common with other rule-based languages. In this section we compare CHR with two important classes of rule-based languages: production rule systems and term rewriting systems.

1. Solve $\langle [c|A], S_0 \uplus S_1, B, T\rangle_n \rightarrowtail \langle S_1 +\!\!+ A, S_0 \uplus S_1, c \wedge B, T\rangle_n$ where c is a built-in constraint, and $vars(S_0) \subseteq fixed(B)$, where $fixed(B)$ is the set of variables fixed by B.[a] *This reconsiders constraints whose matches might be affected by c.*

---

2. Activate $\langle [c|A], S, B, T\rangle_n \rightarrowtail \langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$ where c is a CHR constraint *(which has never been active).*

---

3. Reactivate $\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : 1|A], S, B, T\rangle_n$ where c is a CHR constraint (re-added to A by **Solve** but not yet active).

---

4. Drop $\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle A, S, B, T\rangle_n$ where $c\#i : j$ is an occurrenced active constraint and there is no such occurrence j in P (all existing ones have already been tried thanks to transition 7).

---

5. Simplify $\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail$
$\langle C +\!\!+ A, H_1 \uplus S, \theta \wedge B, T'\rangle_n$ where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H_1' \setminus H_2', d, H_3' \iff g \mid C$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$, and $\mathcal{D}_b \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$. Let $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$ and $T' \neq T$.

---

6. Propagate $\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail$
$\langle C +\!\!+ [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, \theta \wedge B, T'\rangle_n$ where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule in P is

$$r \ @ \ H_1', d, H_2' \setminus H_3' \iff g \mid C$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$,and $\mathcal{D}_b \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$. Let $T' = T \cup \{id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r]\}$ and $T' \neq T$.

---

7. Default $\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : j+1|A], S, B, T\rangle_n$ if the current state cannot fire any other transition.

[a] $v \in fixed(B)$ if $\mathcal{D}_b \models \bar{\exists}_v(B) \wedge \bar{\exists}_{\rho(v)}\rho(B) \rightarrow v = \rho(v)$ for arbitrary renaming ρ.

Figure 2.2: The transition rules of the refined operational semantics ω_r

Transition	Rule	A	S	n
		$[\texttt{gcd}(6), \texttt{gcd}(9)]$	$\emptyset$	1
activate		$[\texttt{gcd}(6)\#1 : 1, \texttt{gcd}(9)]$	$\{\texttt{gcd}(6)\#1\}$	2
default		$[\texttt{gcd}(6)\#1 : 2, \texttt{gcd}(9)]$	$\{\texttt{gcd}(6)\#1\}$	2
default		$[\texttt{gcd}(6)\#1 : 3, \texttt{gcd}(9)]$	$\{\texttt{gcd}(6)\#1\}$	2
default		$[\texttt{gcd}(6)\#1 : 4, \texttt{gcd}(9)]$	$\{\texttt{gcd}(6)\#1\}$	2
drop		$[\texttt{gcd}(9)]$	$\{\texttt{gcd}(6)\#1\}$	2
activate		$[\texttt{gcd}(9)\#2 : 1]$	$\{\texttt{gcd}(6)\#1, \texttt{gcd}(9)\#2\}$	3
default		$[\texttt{gcd}(9)\#2 : 2]$	$\{\texttt{gcd}(6)\#1, \texttt{gcd}(9)\#2\}$	3
simplify	gcd2	$[K \texttt{ is } 9 - 6, \texttt{gcd}(K)]$	$\{\texttt{gcd}(6)\#1\}$	3
solve		$[\texttt{gcd}(3)]$	$\{\texttt{gcd}(6)\#1\}$	3
activate		$[\texttt{gcd}(3)\#3 : 1]$	$\{\texttt{gcd}(6)\#1, \texttt{gcd}(3)\#3\}$	4
default		$[\texttt{gcd}(3)\#3 : 2]$	$\{\texttt{gcd}(6)\#1, \texttt{gcd}(3)\#3\}$	4
default		$[\texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(6)\#1, \texttt{gcd}(3)\#3\}$	4
propagate	gcd2	$[K \texttt{ is } 6 - 3, \texttt{gcd}(K), \texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3\}$	4
solve		$[\texttt{gcd}(3), \texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3\}$	4
activate		$[\texttt{gcd}(3)\#4 : 1, \texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3, \texttt{gcd}(3)\#4\}$	5
default		$[\texttt{gcd}(3)\#4 : 2, \texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3, \texttt{gcd}(3)\#4\}$	5
simplify	gcd2	$[K \texttt{ is } 3 - 3, \texttt{gcd}(K), \texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3\}$	5
solve		$[\texttt{gcd}(0), \texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3\}$	5
activate		$[\texttt{gcd}(0)\#0 : 1, \texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3, \texttt{gcd}(0)\#5\}$	6
simplify	gcd1	$[\texttt{gcd}(3)\#3 : 3]$	$\{\texttt{gcd}(3)\#3\}$	6
default		$[\texttt{gcd}(3)\#3 : 4]$	$\{\texttt{gcd}(3)\#3\}$	6
drop		$\Box$	$\{\texttt{gcd}(3)\#3\}$	6

Table 2.3: Example derivation under the ω_r semantics

2.5.1 Production Rule Systems

The programming paradigm of *production rule systems* originates from Artificial Intelligence research and are mainly used for expert systems (Davis, Buchanan, and Shortliffe 1984). The idea behind it is to represent knowledge as a set of rules of the kind 'if ... then ...'. The if-clause consists of conditions and the then-clause of a sequence of actions. The conditions of the if-clause contain *expression matchings*. Variables that are established in the matchings may be used in the actions of the then-clause.

The *working memory* contains a number of temporary expressions. During the execution of the production rule system, all production rules are identified whose if-clause can be satisfied by some subset of the working memory. Next, the then-clauses of one or more (typically just one) of the identified rules are executed. Typically, actions are either the addition and the removal (sometimes also the updating) of temporary expressions in the working memory. This cycle of identification and execution of actions is repeated until no more production rule is identified.

A few examples of production rule systems are CLIPS (Savely et al. 2005) for C and JRules (ILOG 2004) and Jess (Friendman-Hill 2003) for Java.

CHR as a Production Rule System CHR can be seen as a form of a production rule system specialized for constraint-based host languages. The CHR constraint store corresponds with the working memory and the CHR constraints with the expressions. The if-clause of a CHR rule consists of the matching of the head constraints and the guard. The actions of the then-clause are partially encoded in the head and type of the rule and partially in the body. The removed head constraints correspond with removal actions and the constraints in the body correspond with addition actions. Typically in production rule systems such actions are explicitly stated in the bodies of rules.

In the French Manifico project the use of CHR for typical production rule applications, i.e. knowledge representation, is being researched. Its strong ties to constraint logic programming though and more formal aspects such as the declarative semantics introduced in Section 4.2 make CHR more suitable than other production rule systems for the implementation of constraint solvers and general purpose applications in the context of constraint logic programming languages. The **Solve** transition in the refined operational semantics is rather specific for CHR. This transition reconsiders CHR constraints for rule applications when a built-in constraint is added. In implementations of production rule systems, e.g. the RETE algorithm (Forgy 1982), expressions are usually considered only once actively and there is no notion of triggering.

2.5.2 Term Rewriting Systems

Term Rewriting Systems (TRS) originate from mathematical logic and its theory stems from the study of Church's λ-calculus and Curry's Combinatory Logic. The first use of a TRS as a programming language is (Gorn 1965). See (Dershowitz 1993) for an introduction to rewriting systems. Rewriting rules are directed equations. An initial term is rewritten until a *normal form* is reached, i.e. a form that cannot be rewritten further. Computation consists of repeatedly replacing a (sub)term with a (sub)term that is equal under the directed equations. A noteworthy extension of term rewriting systems are conditional term rewriting systems. Conditional term rewriting systems (Brand, Darringer, and Joyner 1978) consist of guarded rewriting rules. The guard of a rewriting rule consists of a conjunction of equality constraints. An equality constraint holds if the left-hand and the right-hand side have the same normal form.

Other variants of term rewriting systems rewrite structured data instead of syntactic terms. For example, graph rewriting systems (Ehrig 1979) transform graphs.

The following are implementations of TRSs: OBJ3 (Goguen and Malcolm 1996; Goguen, Winkler, Meseguer, Futatsugi, and Jouannaud 1993), Maude (Clavel, Durán, Eker, Lincoln, Martí-Oliet, Meseguer, and Talcott 2003) and Stratego (Visser 2001). Typical applications are theorem proving and modelling but many general purpose applications exist as well. The Stratego language for instance is mainly meant for program transformations.

CHR versus Term Rewriting Systems Many parallels can be drawn between CHR and (conditional) TRSs. Many aspects are similar, but different due to a focus on constraint theory on the one hand and equalities on the other hand:

- Both systems repeatedly rewrite a state until a normal form is reached. While TRSs perform rewritings based on equations, CHR performs rewritings based on a logical (constraint) theory (see Section 4.2).

- The term in the TRS more or less corresponds with the CHR constraint store and subterms with CHR constraints. The TRS term has more explicit structure than the CHR constraint store though: all subterms have a particular position in the term whereas the CHR constraint store is a set without further structure. The same difference exists on the level of the rules. TRS rules rewrite a single term to another one, while several CHR constraints may be removed and added by a single CHR rule.

- Both systems have a means to make applicability of rules conditional. The conditions of conditional TRSs are based on rewriting of equations, while CHR guards are based on the logical constraint theory of the built-in constraints.

Several theoretical results have been ported from term rewriting systems to CHR. For instance, termination conditions of TRSs have been ported from TRSs to CHR (Frühwirth 2000) although they cannot cope with propagation rules. Propagation rules do not exist in TRSs: it is impossible to add something to a term without changing it. Also the notion of confluence and critical pairs (see Section 4.4) is one that has been copied from TRSs to CHR.

While CHR is well integrated with its host language, TRSs are typically stand-alone systems. Also the notion of triggering in the **Solve** rule of the refined operational semantics of CHR is inexistent in TRSs. In TRSs a rewrite rule either does or does not apply to a particular term; there is no notion of updates (the addition of built-in constraints in CHR) that may alter applicability.

Chapter 3

Show Cases of Constraint Handling Rules

3.1 Introduction

In this chapter we illustrate the use of Constraint Handling Rules. We show how three particular applications are modeled and implemented using CHR. The first application, covered in Section 3.2, is an implementation of the classic union-find algorithm. The second application, in Section 3.3, is a framework for testing new memory models for Java. The last application, in Section 3.4, computes the well-founded semantics for general logic programs. Section 3.5 concludes this chapter.

3.2 The Union-Find Algorithm

3.2.1 Introduction

When a new programming language is introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. For example, one often hears the argument that in Prolog some graph algorithms cannot be implemented with best known complexity because Prolog lacks destructive assignment that is needed for efficient update of the graph data structures. In particular, it is not clear whether the union-find algorithm can be implemented with optimal complexity in pure (i.e. side-effect-free) Prolog (Ganzinger and McAllester 2001).

In this text, we will give a positive answer for the Constraint Handling Rules (CHR) programming language. In this section, we show how the algorithm can be implemented elegantly. The actual study of our program's time complexity is done in Section 4.7 after we have covered the necessary time complexity aspects.

Closest to our work is the presentation of a logical algorithm for the union-find problem in (Ganzinger and McAllester 2001). In a hypothetical bottom-up inference rule language with permanent deletions and rule priorities, a set of rules for union-find is given that is proven to run in $\mathcal{O}(M + Nlog(N))$ worst-case time for a sequence of M operations on N elements. The direct efficient implementation of this inference rule system seems not feasible. It is also not clear whether the rules given in (Ganzinger and McAllester 2001) describe the standard union-find algorithm as can be found in text books such as (Cormen, Leiserson, and Rivest 1990). The authors remark that giving a rule set with optimal amortized complexity is complicated.

This section is structured as follows. In the next subsection, we review the classical union-find algorithms. Then, in Section 3.2.3 we present the implementation of the classical union-find algorithm in CHR. An improved version of the implementation, featuring path compression and union-by-rank, is presented next in Section 3.2.4. Finally, Section 3.2.5 concludes.

3.2.2 The Union-Find Algorithm

The classical union-find, also called disjoint set union, algorithm was introduced by Tarjan in the seventies (Tarjan and van Leeuwen 1984). A classic survey on the topic is (Galil and Italiano 1991). The algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations on the sets:

- `make(X)`: create a new set with the single element `X`.
- `find(X)`: return the representative of the set in which `X` is contained.
- `union(X,Y)`: join the two sets that contain `X` and `Y`, respectively (possibly destroying the old sets and changing the representative).

A new element must be introduced exactly once with `make` before being subject to `union` and `find` operations.

In the naive algorithm, these three operations are implemented as follows.

- `make(X)`: generate a new tree with the only node `X`, i.e. `X` is the root.
- `find(X)`: follow the path from the node `X` to the root of the tree. Return the root as representative.
- `union(X,Y)`: find the representatives of `X` and `Y`, respectively. To join the two trees, it suffices to `link` them by making one root point to the other root.

The following imperative pseudo-code implements this algorithm:

Listing 3.1 - The Naive Union-Find Algorithm

```
make(x)
  p[x] ← x

union(x,y)
  link(find(x),find(y))

link(x,y)
   if x ≠ y
         then p[y] ← x

find(x)
   if x ≠ p[x]
         then  return find(p[x])
         else  return x
```

In this pseudo-code `p[x]` denotes the ancestor of `x` in the tree. If `x` is the root, then `p[x]` equals `x`.

The naive algorithm requires $\mathcal{O}(N)$ time per `find` (and `union`) in the worst case, where N is the number of elements (`make` operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve quasi-constant (i.e. almost constant) *amortized* running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root. After `find(X)` returned the root of the tree, we make every node on the path from `X` to the root point directly to the root. The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth. If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one.

For each optimization alone and for using both of them together, the worst case time complexity for a single find or union operation is $\mathcal{O}(log(N))$. For a sequence of M operations on N elements, the worst complexity is $\mathcal{O}(M + Nlog(N))$. When both optimizations are used, the amortized complexity is quasi-linear, $\mathcal{O}(M + N\alpha(N))$, where $\alpha(N)$ is an inverse of the Ackermann function and is less than 5 for all practical N.

In the naive pseudo-code, the `make`, `link` and `find` operations have to be redefined as follows, to add union-by-rank and path compression.

Listing 3.2 - Union-Find with Union-by-Rank and Path Compression

```
make(x)
  p[x] ← x
  rank[x] ← 0

link(x,y)
   if x ≠ y
     if rank[x] ≥ rank[y]
       then p[y] ← x
            rank[x] ← max(rank[x],rank[y] + 1)
       else p[x] ← y

find(x)
   if x ≠ p[x]
         then p[x] ← find(p[x])
   return p[x]
```

The union-find algorithm has applications in graph theory (e.g. efficient computation of spanning trees). By definition of set operations, a union operator working on representatives of sets is an equivalence relation, i.e. we can view sets as equivalence classes. When the union-find algorithm is extended to deal with nested terms to perform congruence closure, the algorithm can be used for term unification in theorem provers and in Prolog.[1] The WAM (Warren 1983), Prolog's traditional abstract machine, uses the basic version of union-find for variable aliasing. While *variable shunting*, a limited form of path compression, is used in some Prolog implementations (Sahlin and Carlsson 1991), we do not know of any implementation of the optimized union-find that keeps track of ranks or other weights.

3.2.3 Implementing Union-Find in CHR

The following CHR program implements the operations and data structures of the naive union-find algorithm without optimizations.

Listing 3.3 - The Naive Union-Find Program

```
make     @ make(X) <=> root(X).

union    @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).
```

[1] It is straightforward to combine the existing CHR solvers for term unification with our union-find implementation

```
findNode @ X ~> PX \ find(X,R) <=> find(PX,R).
findRoot @ root(X) \ find(X,R) <=> R=X.

linkEq   @ link(X,X) <=> true.
link     @ link(X,Y), root(X), root(Y) <=> Y ~> X, root(X).
```

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the operations, so we call them *operation constraints*. The constraints `root/1` and `~>/2` represent the tree data structure and we call them *data constraints*.

The elements we apply `union` to are constants as usual for union-find algorithms. Hence the arguments of all constraints are constants, with the exception of the second argument of `find/2` that must be a variable that is bound to a constant in the rule `findRoot`.

Actually, the use of the built-in constraint `=` in this rule is restricted to returning the element `X` in the parameter `R`. In particular no full unification is ever performed (that could rely on union-find itself).

3.2.4 Optimized Union-Find

The following CHR program implements the optimized classical union-find algorithm with path compression for find and union-by-rank (Tarjan and van Leeuwen 1984). The `union/2` constraint is implemented exactly as for the naive algorithm.

Listing 3.4 - The Optimal Union-Find Program

```
make      @ make(X) <=> root(X,0).

findNode  @ X ~> PX , find(X,R) <=> find(PX,R), X ~> R.
findRoot  @ root(X,_) \ find(X,R) <=> R=X.

linkEq    @ link(X,X) <=> true.
linkLeft  @ link(X,Y), root(X,RX) root(Y,RY) <=> RX >= RY |
              Y ~> X, NRX is max(RX,RY+1), root(X,NRX).
linkRight @ link(X,Y), root(Y,RY), root(X,RX) <=> RY >= RX |
              X ~> Y, NRY is max(RY,RX+1), root(Y,NRY).
```

When compared to the naive version, we see that `root` has been extended with a second argument that holds the rank of the root node. The rule `findNode` has been extended for path compression already during the first pass along the path to the root of the tree. This is achieved by the help of the variable `R` that serves as a place holder for the result of the find operation. The `link` rule has been split

into two rules, `linkLeft` and `linkRight`, to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is updated (both rules are applicable) and the rank is incremented by one.

3.2.5 Union-Find Conclusion

We have shown in this section that it is possible to implement the classical union-find algorithm concisely in CHR. The implementation is extended with two optimizations: path-compression and union-by-rank.

In future work we intend to investigate implementations for other variants of the union-find algorithm. For a parallel version of the union-find algorithm parallel operational semantics of CHR has to be investigated. A dynamic version of the algorithm, e.g. where unions can be undone, would presumably benefit from dynamic CHR constraints as defined in (Wolf 2001; Wolf 2005).

3.3 JmmSolve: a generative Java Memory Model

3.3.1 Introduction

This section covers our implementation in the context of the JSR-133 request (Pugh) for a new memory model for Java. First, Section 3.3.2 briefly introduces the Java memory model and the reason for the formulation of JSR-133 years ago. Next, Section 3.3.3 presents the Concurrent Constraint-Based Memory Machines proposal that was developed by Vijay Saraswat (Saraswat 2004). Then Section 3.3.4 covers the part where we come in: JmmSolve is our generative implementation in Prolog and CHR of Saraswat's proposal, which allows for easy experimentation with the proposal. Finally, Section 3.3.5 concludes.

3.3.2 The Java Memory Model

Memory Models The memory model of a programming language specifies the interaction of multiple threads with main memory. Main memory provides memory locations containing values. A thread is able to interact with these memory locations in several different ways:

- A read operation reads the value in the memory location.
- A write operation replaces the value in the memory location with a new one.
- A lock operation delays other threads' interaction with the memory location until the lock is released.
- An unlock operation releases the lock.

By working on shared memory locations, threads are capable of communicating.

Basically, the memory model says for every value obtained by a read operation in a program by what write operation it has been produced. In a multi-threaded unsynchronized program there need not be a linking from reads to writes with the properties we expect from a single-threaded program: uniqueness of the linking and a logical total ordering of operations.

An *order model* is an important part of a memory model. It specifies in what way memory operations from different threads can and cannot be ordered. For example, the *Sequential consistency* model imposes a total order that only allows interleaved sequentialization of instructions from different threads, while the *Happens Before consistency* model only imposes a total order on events within one thread.

Java Java (Gosling, Joy, and Steele 1996) is currently one of the most popular object-oriented programming languages. Part of its appeal stems from its portability: Java is a multi-platform language in which one can write and compile a program on one platform and run the generated bytecode on many platforms.

For single platform languages certain parts of the language, such as its memory model, are often left unspecified; they are implicitly determined by the platform and programmers familiar with the platform will expect its implicit behavior. However, for a multi-platform language such as Java a full specification of the language, and the memory model in particular, is essential to guarantee portability of programs. If a Java programmer relies on the implicit memory model of the platform she is currently working on, her program may show unexpected behavior on a different platform.

For this reason, the Java language has always had an explicitly specified memory model. However, several undesirable and unexpected properties of the first Java Memory Model (JMM) have come up: it is an unintuitive model that is not easily understood by programmers, it gives rise to some unwanted behavior and it is hard to implement on current hardware architectures.

For example, consider the `java.lang.String` class, part of the Java Standard Library, that implements immutable text strings. In fact, under the old JMM, Strings (objects of this class) are not quite so immutable as intuitively expected by programmers. It is possible, during the initialization of a String that is a substring of another String, to observe a different value than the intended one.

Essentially, the String class has three final (i.e. immutable) fields: a character array, an offset into the character array that represents the start of the string and a length. This encoding allows for the character array to be shared among String objects. In particular this sharing happens when a String object obtained as a substring of another String object. Consider this code:

```
String s1 = "/usr/tmp";
String s2 = s1.substring(4);
```

The String `s2` will have an offset of 4 and a length of 4, but will share the character array with String `s1`. Before the String constructor for `s2` is executed in one thread, all fields of `s2` are initialized to their default values: the length and offset to 0 and the character array to `null`. Because the constructor is not synchronized it is then possible under the old JMM for another thread to first only see the setting of the length and array fields in the constructor, i.e. a value of `"/usr"` and later also see the setting of the offset field to 4. The value of the string appears to have changed from `"/usr"` to `"/tmp"` in the other thread. Clearly, this is a highly undesirable property.

Overview Because of these undesirable properties, Java Specification Request 133 (JSR-133) (Pugh) has called for a new JMM that fixes the current problems. A proposal in the context of JSR-133 is the Concurrent Constraint-based Memory Machines (Saraswat 2004) framework by Vijay Saraswat. In Section 3.3.3 we briefly introduce this framework and in Section 3.3.4 we present JMMSOLVE, our reference implementation of this framework. Section 3.3.5 concludes.

3.3.3 Concurrent Constraint-based Memory Machines

The Concurrent Constraint-based Memory Machines (CCMMs) proposal by Vijay Saraswat (Saraswat 2004) is a framework to express and study different declarative memory models. CCMMs are different from other proposals in that they do not express a memory model in terms of imperative operational semantics, but in terms of constraints. This should facilitate reasoning about the model and its properties, e.g. the *no thin-air reads*[2] property is structurally proved in (Saraswat 2004).

CCMMs associate an event program with a source program. The event program is an abstraction that only keeps the relevant information for the memory model. Every statement in a source program corresponds with several events, i.e. read/write/lock/unlock/... operations on a source variable in a particular thread. Together with the listing of these events equality constraints are imposed over values read and written, e.g. for an assignment of an expression to a variable the value written to the variable should be equal to the value of that expression.

CCMMs models main memory as a constraint store that processes the events with their constraints. Events are processed in batches called *action sets*. Every thread can contribute any number of events with corresponding constraints to an action set. Such an action set is added as a whole to the store. The addition takes care of the following:

- Events in the action set are ordered with respect to events already present in the store.

[2] A thin-air read is a read operation on a variable that returns a value that has not been written to that variable by any thread.

- Events in the action set are ordered with respect to other events in the action set, but in a different thread.
- Read operations are linked to write operations.

The above three steps depend on the particular rules of the underlying ordering model. For example, the Sequential Consistency model only allows interleaved sequentialization of instructions, while the Happens Before model is more relaxed and imposes less ordering.

Example 3.1 Let us consider the example source program of Listing 3.5. The source program is written in a minimalist form of Java source code syntax. Global variables, shared among different threads, are initialized with `init` statements. The code to be executed by a thread is written in a `thread` compound statement and different threads are separated by the `|` symbol. Statements executed by threads are either assignments or synchronization statements such as `lock` and `unlock`. The statements involve global variables, thread-local variables (also called registers; denoted by `r1`, `r2`, ...) and integer values.

```
init x = 0;
init y = 0;
thread { r1 = x;
         y  = 1;
} |
thread { r2 = y;
         x  = 1;
}
```

Listing 3.5: JmmSolve Example Source Program

The event program of this source program is given in Listing 3.6. It consists of two action sets: one for the initialization `t0`, one for the first thread `t1` and the second thread `t2`. The events are named `e1`,...,`e8` and correspond with the different `read` and `write` operations acting on particular memory locations. The equality constraints specify the integer values written in write operations or the relations between reads and writes in the same thread. The ordering constraint `<<` specifies that a one event occurs before another.

A particular instantiation of CCMMs will specify how to add more interthread ordering constraints, namely according to the used order model, and how to link read and write operations, i.e. equality constraints between the values of read and write events.

An example of an additional ordering and equality constraints respecting Sequential Consistency are given in Listing 3.7. The given constraints will cause the

```
% action set for initialization
  write(t0,x,e1), v(e1) = 0,
  write(t0,y,e2), v(e2) = 0,
  e1 << e2

% action set for threads
  read(t1,x,e3), write(t1,r1,e4), v(e3) = v(e4),
  write(t1,y,e5), v(e5) = 1,
  e3 << e4 << e5

  read(t2,y,e6), write(t2,r2,e7), v(e6) = v(e7),
  write(t2,x,e8), v(e8) = 1,
  e6 << e7 << e8
```

Listing 3.6: JMMSOLVE Example Event Program

final values of `r1` and `r2` to be `1` and `1` respectively. Other final values are possible in weaker order models.

```
  e2 << e3,
  e5 << e6,
  v(e3) = v(e1),
  v(e6) = v(e5)
```

Listing 3.7: JMMSOLVE Example Constraints To Satifsy Sequential Consistency

---

In addition to the requirements of JSR-133, CCMMs has set itself the requirement to be generative. This means that given a program it should be possible to generate all valid executions, in particular all valid linkings of reads to writes. It is the goal of JMMSOLVE to prove this point by providing exactly such a generative implementation of CCMMs.

3.3.4 The JMMSOLVE Implementation

Because of the declarative constraint-based nature of CCMMs we have chosen constraint logic programming (see Section 2.3) as the technology for this generative implementation. In particular, JMMSOLVE has been implemented in SWI-Prolog (Wielemaker 2004), with its new support for CHR (see Section 6.4.2).

The current working of JMMSOLVE is as follows. A source program (in a simplified syntax) is read in and converted to an event program with constraints.

The event program is partitioned into one action set for the initialization and one for all the threads. Both action sets are added to an empty store together with the necessary ordering constraints according to the memory model. Finally all valid linkings from reads to writes are generated for each action set.

The compiler from source to event programs is rather straightforward, using definite clause grammars (DCGs). On the other hand, the generative part of JmmSolve is more complicated and uses a mix of ordinary Prolog and CHR constraints. CHR constraints are used in particular:

- For the event ordering constraint `(<<)/2` together with the ordering and linking rules of the memory model. As the ordering constraint is subject to order model-specific rules, this definitely calls for CHR. Indeed, it has proven to be rather easy to translate the rules of order models into CHR rules.

  Listing 3.8 presents the basic rules for the `<</2` constraint.

```
antireflexive @ E  << E             <=> fail.
antisymmetric @ Ex << Ey,  Ey << Ex <=> fail.
redundant     @ Ex << Ey \ Ex << Ey <=> true.
transitive    @ Ex << Ey,  Ey << Ez ==> Ex << Ez
```

Listing 3.8: The JmmSolve `<</2` Order Constraint

- For a minimalist integer constraint solver with constraint implication and equality. This only serves as a proof of concept. We could have used just as well another full blown integer solver. However, this observation was not clear at the beginning and CHR has allowed us to go ahead without worrying over possible interoperability problems. Moreover, no integer solver was available for SWI-Prolog at that time. Now that the prototype has been established, it is fairly easy to indeed improve efficiency and implement a genuine integer solver. However, the performance of integer constraint solving seems not critical in JmmSolve.

  Sample code is given in Listing 3.9. The `expression(V,E)` constraint constraints the value `V` to be equal to the result of arithmetic expression `E`. The constraint is simply delayed until the expression is ground and hence can be evaluated in the usual way. Similarly, the constraint `eq(X,Y,T)` delays until `X` and `Y` are definitely equal or different, and then binds `T` to `1` or `0` respectively. This boolean value is used by the `ite(V,C,TV,EV)` constraint to conditionally assign either of two values to a variable `V`. Such a construct is necessary to cope with conditional assignments in the source language.

We do exploit the actual operational semantics of CHR for tasks which may be considered impure in traditional constraint logic programming, such as collecting

```
equality @ expression(V,E) <=> ground(E) | V is E.

reified_eq_true  @ eq(X,X,T) <=> T = 1.
reified_eq_false @ @ eq(X,Y,T) <=>
                      number(X), number(Y), X \== Y | T = 0.

ite_then   @ ite(V,1,T,_) <=> V = T.
ite_else   @ ite(V,0,_,E) <=> V = E.
ite_either @ ite(V,_,B,B) <=> V = B.
```

Listing 3.9: The JmmSolve `expression/2` Integer Constraint

from the store all constraint of a particular kind and relying on the order in which constraints are added to the program. However, some of these tasks are necessary for interfacing with the Prolog code and they certainly facilitate the programming.

3.3.5 JmmSolve Conclusion

In this section we have shown with JmmSolve that the CCMMs proposal of Vijay Saraswat can actually be implemented. While implementing JmmSolve in Prolog, CHR has proven to be a valuable *programmer's tool* next to traditional ones such as DCGs. It has allowed the elegant expression of order model rules and the rapid and simple implementation of the necessary integer constraint solver functionality. If the performance of the latter's efficiency is not adequate, it can easily be replaced with a more involved and more efficient implementation. However, the combinatorial explosion of possible orderings and linkings, rather than the integer constraint solving, seems to be the bottleneck.

The current implementation of JmmSolve is available for download at `http://www.cs.kuleuven.ac.be/~toms/jmmsolve/` and contains rules for the Happens Before model. This work was presented as a poster at the 2004 International Conference of Logic Programming (Schrijvers 2004).

Unfortunately, Vijay Saraswat has formulated his CCMMs proposal only months before the deadline of JSR133 regarding the decision on the new memory model for Java 5. This limited time frame has proven to be insufficient to raise much interest in the JMM community. The elected memory model, formulated in (Pugh et al. 2004), is strictly imperative in nature and was proposed much earlier by some of the leading people in the JMM community.

3.4 A Well-Founded Semantics Algorithm

3.4.1 Well-Founded Semantics

As another illustration of the power of CHR for general purpose applications, we have implemented an algorithm that computes the *well-founded* semantics of *simple* general logic programs. It is based on the definition of the well-founded semantics for general logic programs by Van Gelder, Ross and Schlipf (Van Gelder, Ross, and Schlipf 1991). The well-founded semantics are a formal declarative semantics of Horn rule logic programs extended with negative subgoals, called *general logic programs*. The semantics are intended to be natural and intuitive.

In (Van Gelder, Ross, and Schlipf 1991) general logic programs are defined as follows.

Definition 3.1 *A general logic program* **P** *is a finite set of general rules, which may have both positive and negative subgoals. A general rule is written with its head, or conclusion on the left, and its subgoals (body), if any to the right of the symbol ":-", which may be read "if". For example,*

$$p(X) \ :\!- \ a(X), \textbf{\textit{not}}\ b(X).$$

is a rule in which $p(X)$ is the head, $a(X)$ is a positive subgoal, and $b(X)$ is a negative subgoal. This rule may be read as "$p(X)$ if $a(X)$ and not $b(X)$." A Horn rule is one with no negative subgoals, and a Horn logic program is one with only Horn rules.

The well-founded semantics associates a truth value with all possible ground positive and negative literals: heads or subgoals with arguments constructed from any function symbols and constants in the program. The possible truth values are *true*, *false* and *undefined*.

The number of possible ground literals that can be constructed from a function-free program is finite and arguments consist of atoms only. This restriction allows for a finite number of truth values associated with the program: one for each distinct ground literal. Hence the program can be grounded first before computing the model: all rules are replaced by all possible ground rules obtained by replacing non-ground literals with corresponding ground ones. As every distinct ground literal can be replaced by a unique new argument-free literal, it is even not necessary to consider programs with arguments in them. Hence, for the purpose of this text, we restrict ourselves to argument-free general logic programs: the literals have no arguments, i.e. they are positive and negative atoms. For such a program **P** the *Herbrand Base* BASE(**P**) is simply the set of all atoms appearing in the program.

In order to properly define the well-founded semantics, we first have to introduce a number of auxiliary definitions, taken from (Van Gelder, Ross, and Schlipf 1991).

Definition 3.2 *For a set of literals S we denote the set formed by taking the complement of each literal in S by $\neg \cdot S$.*

- *We say literal q is* inconsistent *with S if $q \in \neg \cdot S$.*
- *Sets of literals R and S are* inconsistent *if some literal in R is inconsistent with S, i.e. if*

$$R \cap \neg \cdot S \neq \emptyset$$

A set of literals is inconsistent *if it is inconsistent with itself; otherwise it is* consistent*.*

We represent the subset of all positive literals in S by S^+ and the set of all negative literals in S by S^-.

Of special interest are particular sets of literals: partial interpretations.

Definition 3.3 (Partial Interpretation) *Given a program $\mathbf{P}$, a* partial interpretation *I is a consistent set of literals whose atoms are in the Herbrand base of $\mathbf{P}$. We say a literal is* true in *I when it is in I and say it is* false in *I when its complement is in I.*

Definition 3.4 (Unfounded Set) *Let a program $\mathbf{P}$ and a partial interpretation I be given. We say A is an* unfounded set *of $\mathbf{P}$* with respect to *I if each atom $p \in A$ satisfies the following condition: For each instantiated rule R of $\mathbf{P}$ whose head is p, at least one of the following holds:*

1. *Some positive or negative subgoal q of the body is false in I.*
2. *Some positive subgoal of the body occurs in A.*

The well-founded semantics is defined in terms of transformations of partial interpretations.

Definition 3.5 *Transformations $\mathbf{T}_P$, $\mathbf{U}_P$ and $\mathbf{W}_P$ are defined as follows:*

- *$p \in \mathbf{T}_P(I)$ if and only if there is some instantiated rule R of $\mathbf{P}$ such that R has head p, and each subgoal literal in the body of R is true in I.*
- *$\mathbf{U}_P(I)$ is the greatest unfounded set of $\mathbf{P}$ with respect to I.*
- *$\mathbf{W}_P(I) = \mathbf{T}_P(I) \cup \neg \cdot \mathbf{U}_P(I)$.*

The actual definition of the well-founded semantics is then as follows.

Definition 3.6 (Well-Founded Semantics) *The* well-founded semantics *$\mathbf{W}_P^*$ of a program $\mathbf{P}$ is the "meaning" represented by the least fixed point of $\mathbf{W}_P$ starting from $\emptyset$; every positive literal denotes that its atom is true, every negative literal denotes that its atom is false, and missing atoms have no truth value assigned by the semantics.*

3.4.2 The Algorithm

Many algorithms exist for the computation of the well-founded semantics and much work has been done to improve the efficiency of these algorithms, e.g. (Berman, Schlipf, and Franco 1995) and (Brass, Dix, Freitag, and Zukowski 1998).

We have implemented an algorithm that was developed by Simons (Simons 2000) as part of `smodels`, an algorithm to compute the stable semantics of logic programs.

This algorithm corresponds with the *expand* function of `smodels` (Simons 2000):

$$\mathbf{W}_P^* = expand(P, \emptyset)$$

```
function expand(P, I)
  repeat
    I' := I
    I := Atleast(P, I)
    I := I ∪ {¬p | p ∈ Base(P) ∧ p ∉ Atmost(P, I)}
  until I' = I .
```

Listing 3.10: The *expand* Function

The *expand* function is defined in Listing 3.10. Here the outer **repeat** loop corresponds with a least fixedpoint computation, the *Atleast* function corresponds with the least fixedpoint of $\mathbf{\Phi}_P(I)$ and the *Atmost* function with the greatest fixedpoint of $\mathbf{T}_P(I)$.

The Fitting operator $\mathbf{\Phi}_P$ is defined as follows.

Definition 3.7 (Fitting Operator) *The Fitting operator $\mathbf{\Phi}_P$ is defined as follows:*

- $p \in \mathbf{\Phi}_P(I)$ *if and only if there is some rule R of $\mathbf{P}$ such that R has head p, and each subgoal literal in the body of R is true in I.*
- $\neg p \in \mathbf{\Phi}_P(I)$ *if and only if for every rule R of $\mathbf{P}$ such that R has head p, at least one subgoal in the body of R is false in I.*

We now proceed with explaining the *Atleast* and *Atmost* functions in more detail at the hand of their respective implementation in imperative pseudo-code. The contrast with the corresponding CHR code, in Section 3.4.3, will be apparent.

At Least The function *Atleast* is computed by the *atleast*() procedure that makes use of the following variables. The subscript l indicates that these are related to *Atleast*, to avoid confusion with similarly named variables for *Atmost*.

$p.headof_l$ The number of active rules whose head contains p.

$p.inI_l^+$ A flag that is true if p is in the current interpretation.

$p.inI_l^-$ A flag that is true if $\neg p$ is in the current interpretation.

$p.plist_l$ The rules in whose bodies p appears.

$p.nlist_l$ The rules in whose bodies $\neg p$ appears.

$r.literal_l$ The number of literals in the body of r that are not in the current interpretation.

$r.inactive_l$ The number of literals in the body of r that are in $\neg \cdot I$.

$r.body_l$ The body of the rule r.

$r.head_l$ The head of the rule r.

These variables are initialized properly at the start of the algorithm.

Two queue-like data structures are used in *atleast*(): $posq_l$ and $negq_l$. We assume that they are implemented such that pushing an atom onto the end of a queue does nothing if the atom is already on the queue. In addition, we assume that pushing an atom p whose $p.inI_l^+$ flag is true onto the queue $posq_l$ or pushing an atom p whose $p.inI_l^-$ flag is true onto the queue $negq_l$ has no effect

The procedure *atleast*() is then defined as in Listing 3.11 with auxiliary procedures given in Listing 3.12.[3] Before computing $Atleast(P, I)$ the queues have to be initialized: $posq_l$ with I^+ and the heads of the rules whose bodies are empty, and $negq_l$ with the atoms in I^-. The procedure can also be used incrementally. If $Atleast(P, I)$ has been computed and we are going to compute $Atleast(P, I \cup \{p\})$, then we just initialize $posq_l$ with p and call *atleast*().

At Most The function *Atmost* is computed by the *atmost*() procedure that makes use of the following variables. The subscript m indicates that these are related to *Atmost*.

$p.inI_m^+$ A flag that is true if p is in the current interpretation.

$p.plist_m$ The rules in whose bodies p appears.

$r.pliteral_m$ The number of positive literals in the body of r that are not in the current interpretation.

$r.head_m$ The head of the rule r.

[3] Only the code relevant for the well-founded semantics is given.

```
procedure atleast()
  while posq_l or negq_l are not empty do
    if posq_l is not empty then
      p := posq_l.pop()
      p.inI_l^+ := true
      for each rule r ∈ p.plist_l do
        r.fire_l()
      end for
      for each rule r ∈ p.nlist_l do
        r.inactivate_l()
      end for
    end if
    if negq_l is not empty then
      p := negq_l.pop()
      p.inI_l^- := true
      for each rule r ∈ p.nlist_l do
        r.fire_l()
      end for
      for each rule r ∈ p.plist_l do
        r.inactivate_l()
      end for
    end if
  end while.
```

Listing 3.11: The Implementation of *Atleast*

```
procedure r.fire_l()
  r.literal_l := r.literal_l − 1
  if r.literal_l = 0 then
    posq_l.push(r.head_l)
  end if.
procedure r.inactivate_l()
  r.inactive_l := r.inactive_l + 1
  if r.inactive_l = 1 then
    p := r.head_l
    p.headof_l := p.headof_l − 1
    if p.headof_l = 0 then
      negq_l.push(p)
    end if
  end if.
```

Listing 3.12: Auxiliary Procedures for *atleast*()

These variables are initialized properly at the start of the algorithm.

The procedure *atmost*() is defined in Listing 3.13 with an auxiliary procedure given in Listing 3.14. For our purpose we have chosen a definition that more closely resembles the definition of *atleast*(). A single queue is used in *atmost*(): $posq_m$. We make similar assumptions about is as for $posq_l$. The procedure can also be used incrementally. If $Atmost(P, I)$ has been computed and we are going to compute $Atmost(P, I \cup \{p\})$, then we just initialize $posq_m$ with p and call *atmost*().

```
procedure atmost()
  while posq_l not empty do
    p := posq_m.pop()
    p.inI_m^+ := true
    for each rule r ∈ p.plist_m do
      r.fire_m()
    end for
  end while.
```

Listing 3.13: The Implementation of *Atmost*

```
procedure r.fire_m()
  r.pliteral_m := r.pliteral_m − 1
  if r.pliteral_m = 0 then
    posq_m.push(r.head_m)
  end if.
```

Listing 3.14: Auxiliary Procedure for *atmost*()

3.4.3 The CHR Implementation

The algorithm is quite a challenge for CHR. Not only do the *atleast*() and *atmost*() steps have to be implemented, but also the alternation of the two steps has to be encoded, information must flow between the two steps and a global fixedpoint has to be reached.

The individual functionality of the two procedures *atleast*() and *atmost*() has been implemented without any particular ordering of rules in mind, but to tackle their interoperation, we heavily rely on the actual operational semantics of CHR: the order of the rules is important for the correct execution of the program.

In the following we first explain how the functionality of *atleast*() and *atmost*() is realized. Then we discuss how they are made to interoperate.

At Least The CHR rules that cover the functionality of *atleast*() is shown in Listing 3.15. Some of the variables used by *atleast*() are related to CHR constraints

CHR Constraint	Meaning
`headof1(P,I)`	$p.headof_l = i$
`inIplus1(P)`	$p.inI_l^+ = \text{true}$
`inIminus1(P)`	$p.inI_l^- = \text{true}$
`literal1(R,I)`	$r.literal_l = i$
`inBodyPlus1(P,R)`	$r \in p.plist_l$
`inBodyMinus1(P,R)`	$p \in r.body_l{}^-$
`head1(R,P)`	$r.head_l = p$

Table 3.1: The relation between the *atleast*() variables and CHR constraints

CHR Constraint	Meaning
`inIplus2(P)`	$p.inI_m^+ = \text{true}$
`pliteral2(R,I)`	$r.pliteral_m = i$
`inBodyPlus2(P,R)`	$r \in p.plist_m$
`head2(R,P)`	$r.head_m = p$

Table 3.2: The relation between the *atmost*() variables and CHR constraints

in the code; the relation between them is given in Table 3.1.

The CHR code does not explicitly maintain the queues $posq_l$ and $negq_l$: because of the CHR semantics, remaining work is maintained implicitly. Since the $fire_l()$ and $inactivate_l()$ operations in *atleast*() are mutually confluent, it is even not necessary to deal with them in a particular order.

The `fire_posq` and `fire_negq` cover the two cases where the $fire_l()$ procedure is called. Similarly, the `inactivate_posq` and `inactivate_negq` cover the two cases where $inactivate_l()$ is called. The `fire` and `inactivate` rules implement the unconditional parts of the corresponding procedures. The `all_literals_true` rule implements the conditional part of the $fire_l()$ procedure and the `no_active_rules` rule implements the conditional part of the $inactivate_l()$ rule.

The rules of Listing 3.15 should be preceded by a number of rules (not listed) to remove any redundant constraints listed in Table 3.1. Namely, if $p.inI_l^+$ or $p.inI_l^-$ is known, any information on rules r with $r.head_l = p$ is no longer of interest. This avoids redundant triggering of constraints.

At Most The CHR rules that covers the functionality of *atmost*() is shown in Listing 3.16. The code is similar to that for *atleast*().

Some of the variables used by *atmost*() are related to CHR constraints in the code; the relation between them is given in Table 3.2.

```
fire_posq          @ inIplus1(P) \ inBodyPlus1(P,R) <=>
                        fire1(R).

fire_negq          @ inIminus1(P) \ inBodyMinus1(P,R) <=>
                        fire1(R).

fire               @ fire1(R), literal1(R,I) <=>
                        J is I - 1, literal1(R,J).

all_literals_true @ literal1(R,0), head1(R,P) <=>
                        inIplus1(P).

inactivate_posq    @ inIplus1(P) \ inBodyMinus1(P,R) <=>
                        inactivate1(R).

inactivate_negq    @ inIminus1(P) \ inBodyPlus1(P,R) <=>
                        inactivate1(R).

inactivate         @ inactivate1(R), head1(R,PP),
                     literal1(R,_), headof1(PP,I) <=>

                        J is I - 1, headof1(PP,I).
no_active_rules    @ headof1(P,0) <=>
                        inIminus1(P).
```

Listing 3.15: The CHR Implementation of *atleast*()

```
fire_posq2          @ inIplus2(P) \ inBodyPlus2(P,R) <=>
                        fire2(R).

fire2               @ fire2(R), pliteral2(R,I) <=>
                        J is I - 1, pliteral2(R,J).

all_literals_true2 @ pliteral2(R,0), head2(R,P) <=>
                        inIplus2(P).
```

Listing 3.16: The CHR Implementation of *atmost*()

Glue Code Besides the CHR code for *Atleast* and *Atmost*, some additional glue code is needed to make them interoperate and to compute their fixedpoint. This glue code relies on actual rule order and hence the refined operational semantics.

Firstly, from the partial interpretation I obtained from *atmost*() the set $\{\neg p \mid p \in \text{BASE}(P) \wedge p \notin I\}$ should be computed. For this purpose, an `undefined2(P)` constraint is called for every atom `P` that is undefined in the initial partial interpretation. When `inPlus2(P)` is added, the corresponding `undefined2(P)` is removed. At the end of *atmost*() every remaining `undefined2(P)` constraint is replaced with a `inIminus(P)` constraint.

Secondly, to alternate between the two procedures, the `atmost` and `atleast` constraints are used. The former indicates a transition from computing *atleast*() to *atmost*(), the latter the other way around. These two constraints ensure the information in the two steps is carried over to each other. The `pliteral1/2` constraint, not directly used in *atleast*(), is maintained for this purpose: to carry it over to *atmost*() where it is used.

Finally, to compute the fixedpoint of the two steps, it is checked whether there is a remaining `undefined2(P)` constraint after *atmost*(). If there is, some new information has been derived by this step. Otherwise, no new information is obtained and the algorithm has reached a fixedpoint.

3.4.4 Well-Founded Semantics Conclusion

In this section, we have shown how to implement in CHR an algorithm that computes the well-founded semantics of argument-free general logic programs.

On the one hand, the high-level theoretical semantics of CHR have allowed for a much more elegant and concise formulation of the confluent parts of the algorithm. The CHR code does not contain explicit loop constructs and while the core imperative pseudo-code is about 45 lines long, the CHR code needs only 11 rules.

On the other hand, the operational semantics of CHR have allowed for the implementation of more imperative features such as the alternating steps in the algorithm. The fact that CHR permits these operational kinds of techniques considerably improves its usability and suitability for a wider range of problems. However, more syntactical support for alternating between steps and reuse of rules for similarly behaving constraints would make programming in CHR even more convenient.

The full source code of the CHR program, called `wfs`, is given in Appendix A. It is also available from (Schrijvers 2005).

3.5 Conclusion

In this chapter we have illustrated the use of CHR in three different contexts: the union-find algorithm, a framework for Java memory models and the well-founded semantics. We have introduced the three problem domains, explained their solutions and discussed our implementation in CHR.

We have shown how control constructs, such as choice and alternation, can be implemented under the refined operational semantics. Our programs are elegant, compact and fairly easy to understand.

Chapter 4

Theoretical Properties

4.1 Introduction

In this chapter we present notions and properties of CHR programs that are of a more theoretical nature: the declarative semantics, confluence and time complexity. Some of these properties, the declarative semantics and confluence, may be used for program verification, i.e. to verify whether some of a CHR program's properties correspond with the intended properties. While some could also be exploited for program optimization, most notably confluence, we are not aware of any such application.

Overview Firstly, Section 4.2 discusses the *declarative semantics* of CHR programs. This semantics associates a logical meaning with the operational nature of CHR programs and make CHR a proper declarative language.

Secondly, Section 4.4 covers the notion of *confluence* which establishes whether the non-determinism in the operational semantics of CHR may yield multiple final states from a given initial state, which is often an undesired behavior.

Finally, Section 4.6 briefly introduces the general *time complexity* bound for CHR programs. Currently no general space complexity result exists for CHR.

These properties are each illustrated with an example or a case study on the union-find programs of Section 3.2, which were conducted in cooperation with Thom Frühwirth (Schrijvers and Frühwirth 2005; Schrijvers and Frühwirth 2005; Schrijvers and Frühwirth 2004). Section 4.3 is an illustration of the declarative semantics and Sections 4.5 and 4.7 are case studies of confluence and time complexity respectively. Finally, Section 4.8 summarizes this chapter.

4.2 Declarative Semantics

Besides an operational semantics associated with a CHR program $\mathcal{P}$, also a *declarative semantics* or *logical meaning* is assigned to it. A CHR program $\mathcal{P}$ is a model of a logical theory $[\![\mathcal{P}]\!]$. This theory $[\![\mathcal{P}]\!]$, or logical meaning of the program, is represented by a set of first order logic formulas. The theory needs to be considered in conjunction with the logical theory $\mathcal{D}_b$ of the host language (built-in) constraints.

For each rule in the program a formula is derived. The following table lists the formula for each type of rule (see also (Frühwirth 1998)).

Type	Rule	Formula
simplification	$H \ <=> \ C \mid B.$	$\forall(C \rightarrow (H \leftrightarrow \exists \bar{y}\ B))$
propagation	$H \ ==> \ C \mid B.$	$\forall(C \rightarrow (H \rightarrow \exists \bar{y}\ B))$
simpagation	$H_1 \setminus H_2 \ <=> \ C \mid B.$	$\forall(C \rightarrow (H_1 \rightarrow (H_2 \leftrightarrow \exists \bar{y}\ B)))$

where $\bar{y}$ are the variables that appear only in the body B. $\forall F$ denotes the universal closure of a formula F.

4.2.1 Relation to the Operational Semantics

Not surprisingly there is a close relation between the declarative and the theoretical operational semantics of CHR. The theoretical operational semantics of CHR can be seen as rewriting the execution state while preserving logical equivalence under the logical theory of the program.

More formally, we can assign the following logical formula to an execution state $\sigma = \langle G, S, B, T \rangle_n$ with goal G, CHR constraint store S, built-in constraint store B, propagation history T and next free identifier n:

$$\mathit{meaning}(\sigma) \stackrel{\text{def}}{=} G \wedge \bigwedge \{c \mid c\#i \in S\} \wedge B$$

One can verify that for any transition in the theoretical operational semantics, it holds that:

Lemma 4.1 (Elementary Soundness)

$$\forall \sigma_1, \sigma_2 : (\sigma_1 \rightarrowtail_{\mathcal{P}} \sigma_2) \Longrightarrow ([\![\mathcal{P}]\!], \mathcal{D}_b \models \mathit{meaning}(\sigma_1) \Leftrightarrow \mathit{meaning}(\sigma_2))$$

From Lemma 4.1 the soundness and completeness theorems of (Frühwirth 1998) follow. Paraphrased they boil down to this:

Theorem 4.1 (Soundness) *Let $\mathcal{P}$ be a CHR program and G a goal with $\langle G, \mathit{true}, \mathit{true}, \emptyset \rangle_0 \rightarrowtail^*_{\mathcal{P}} \sigma$ and $C = \mathit{meaning}(\sigma)$, then:*

$$[\![\mathcal{P}]\!], \mathcal{D}_b \models \forall(G \Leftrightarrow C)$$

Theorem 4.2 (Completeness) *Let $\mathcal{P}$ be a CHR program and G a goal for which $\langle G, true, true, \emptyset\rangle_0$ has at least one finite final state and C be a conjunction of constraints. If $[\![\mathcal{P}]\!], \mathcal{D}_b \models \forall(G \Leftrightarrow C)$, then $\langle G, true, true, \emptyset\rangle_0$ has a final state whose meaning is C' for which:*

$$[\![\mathcal{P}]\!], \mathcal{D}_b \models \forall(C \Leftrightarrow C')$$

4.2.2 Scope of the Declarative Semantics

While it is possible to derive a logical theory $[\![\mathcal{P}]\!]$ for any CHR program $\mathcal{P}$, such a logical theory may not be of any use. In particular when a program is written with only the refined operational semantics ω_r in mind, it may have a useful operational meaning while the declarative meaning is not relevant.

Typically the declarative meaning only makes sense when the programmer intends to implement (part of) a particular constraint theory $\mathcal{D}$. The declarative meaning $[\![\mathcal{P}]\!]$ then is a means to verify the implementation $\mathcal{P}$ against the intended logical theory $\mathcal{D}$.

Definition 4.1 (Soundness) *A CHR program $\mathcal{P}$ is sound with respect to logical theory $\mathcal{D}$, if:*

$$\forall F : [\![\mathcal{P}]\!] \models F \quad \Rightarrow \quad \mathcal{D} \models F$$

where F is a syntactically valid formula for the constraint domain.

Soundness is a very important property, as it ascertains that any theorem that can be proven with the program's logical theory can also be proven with the intended logical theory. This establishes that any final state is not only equivalent to the initial state under the program's logical theory $[\![\mathcal{P}]\!]$, but also under the intended logical theory $\mathcal{D}$. Typically soundness is an intended property and unsoundness is an indication of a bug in the program. Hence soundness is an interesting property for program verification. It can be established by showing that every formula F that corresponds with a rule in the program $\mathcal{P}$ holds in the intended theory $\mathcal{D}$. These proofs can be done by hand or with an automated theorem prover, though we are not aware of any automation used so far in this process.

Definition 4.2 (Completeness) *A CHR program $\mathcal{P}$ is complete with respect to logical theory $\mathcal{D}$, if:*

$$\forall F : \mathcal{D} \models F \quad \Rightarrow \quad [\![\mathcal{P}]\!] \models F$$

where F is a syntactically valid formula for the constraint domain.

Completeness establishes that the logical theory of a CHR program completely covers the intended logical theory. It is less often required from a CHR program as it is much more difficult to relate it to operational aspects. Indeed, the fact that a particular equivalence of states holds under the program's logical theory is by

no means a promise that a transition or sequence of transitions from the one state to the other is at all possible under the operational semantics of CHR. Moreover, it may be so that particular equivalences of states are irrelevant for the intended uses of the program.

The class of programs CHR was originally intended for, namely constraint solvers, is one where the declarative semantics are relevant: a constraint solver implements (part of) a logical constraint theory. In Chapter 9 we will look at a particular extension of CHR programs that only makes sense for programs with a meaningful declarative semantics and constraint solvers in particular.

4.3 Declarative Semantics Example: Union-Find

The following logical theory $[\![UF]\!]$ was derived from the naive union-find CHR program of Section 3.2 using the rules of the previous section:

make	$make(A) \Leftrightarrow root(A)$
union	$union(A,B) \Leftrightarrow \exists XY(find(A,X) \wedge find(B,Y) \wedge link(X,Y))$
findNode	$find(A,X) \wedge A{\rightarrow}B \Leftrightarrow find(B,X) \wedge A{\rightarrow}B$
findRoot	$root(A) \wedge find(A,X) \Leftrightarrow root(A) \wedge X{=}A$
linkEq	$link(A,A) \Leftrightarrow true$
link	$link(A,B) \wedge root(A) \wedge root(B) \Leftrightarrow B{\rightarrow}A \wedge root(A)$

From the logical reading of the rule `link` it follows that $B{\rightarrow}A \wedge root(A) \Rightarrow root(B)$, i.e. $root(A)$ holds for every node A in the tree, not only for root nodes. Indeed, we cannot adequately model the update from a root node to a non-root node in first order logic, since first order logic is monotonic: formulas that hold cannot cease to hold. In other words, the `link` rule is where the union-find algorithm is *non-logical* since it requires an update that is destructive in order to make the algorithm efficient.

In the union-find algorithm, by the definition of the set operations, a union operator working on representatives of sets is an equivalence relation observing the usual axioms of the equivalence theory $\mathcal{D}_{=}$:

reflexivity	$union(A,A) \Leftrightarrow true$
symmetry	$union(A,B) \Leftrightarrow union(B,A)$
transitivity	$union(A,B) \wedge union(B,C) \Rightarrow union(A,C)$

To show that these axioms hold for the logical reading of the program, we can use the following observations: since the unary constraints `make` and `root` must

hold for any node in the logical reading, we consider them to be equal to *true*. By the rule `findRoot`, the constraint `find` must be an equivalence. Hence its occurrences can be replaced by `=`. Now `union` is defined in terms of `link`, which is reflexive by rule `linkEq` and logically equivalent to `~>` by rule `link`. But `~>` must be equivalence like `find` because of rule `findNode`. Hence all binary constraints define equivalence. After renaming the constraints accordingly, we obtain the following theory:

union	$A{=}B \Leftrightarrow \exists XY(A{=}X \wedge B{=}Y \wedge X{=}Y)$
findNode	$A{=}X \wedge A{=}B \Leftrightarrow B{=}X \wedge A{=}B$
findRoot	$A{=}X \Leftrightarrow X{=}A$
linkEq	$A{=}A \Leftrightarrow \mathit{true}$
link	$A{=}B \Leftrightarrow B{=}A$

It is easy to see that these formulas are logically equivalent to the axioms for equality. This proves that the union-find program is sound and complete with respect to the intended equivalence theory $\mathcal{D}_{=}$.

4.4 Confluence

Typically, more than one CHR rule is applicable to an execution state. It is obviously a highly desirable property that, no matter what rule is applied, the final result of a derivation is always syntactically the same. This property is called *confluence*.

In this section we summarize the existing work on confluence for CHR. The main confluence result for CHR is (Abdennadher 1997), which covers the entire CHR language. The earlier confluence result (Abdennadher, Frühwirth, and Meuss 1996) was restricted to CHR programs without propagation rules.

In (Abdennadher 1997) it is assumed that every program state can be considered to be in a normal form with respect to the built-in constraints: equivalent built-in constraints are syntactically identical. Moreover, two execution states originating from the same initial state are *variants* of each other when they are syntactically identical modulo renaming of constraint identifiers, renaming of variables not present in the initial state and redundant tuples in the propagation history.

Definition 4.3 *Two execution states σ_1 and σ_2 are called* joinable *if there exist states σ_1' and σ_2' such that $\sigma_1 \rightarrowtail^* \sigma_1'$ and $\sigma_2 \rightarrowtail^* \sigma_2'$ and σ_1' and σ_2' are variants.*

Definition 4.4 *A CHR program $\mathcal{P}$ is called* confluent *if for all execution states $\sigma, \sigma_1, \sigma_2$:*

$$\sigma \rightarrowtail^* \sigma_1 \wedge \sigma \rightarrowtail^* \sigma_2 \Longrightarrow \sigma_1 \text{ and } \sigma_2 \text{ are joinable}$$

Definition 4.5 *A CHR program $\mathcal{P}$ is called* locally confluent *if for all execution states $\sigma, \sigma_1, \sigma_2$:*

$$\sigma \rightarrowtail \sigma_1 \wedge \sigma \rightarrowtail \sigma_2 \Longrightarrow \sigma_1 \text{ and } \sigma_2 \text{ are joinable}$$

It is not possible to check joinability of all derived execution states from a common direct ancestor state, as there is an unbounded number of such ancestor states and hence derived states. However, it is possible to construct a finite number of minimal states in which multiple CHR rules are applicable. These minimal states can be extended to any possible context by adding more constraints.

We further restrict the discussion to non-trivial direct ancestor states where the application of one rule inhibits the application of another rule. A rule may add constraints (its body) and remove constraints (the removed head constraints). If we restrict ourselves to monotonic built-in constraints, the addition of new constraints cannot inhibit the application of a rule.[1] The application of a rule inhibits the application of another rule if the one rule removes at least one constraint needed by the other rule. So the two rules must have overlapping head constraints and at least one of the overlapping constraints must be removed by one rule. The pair of states resulting from this overlap is called a *critical pair*.

Definition 4.6 (Critical Pair) *Let R and R' be rules with respective heads H and H', guards g and g', and bodies B and B'. Either $H_r \neq \Box$ or $H'_r \neq \Box$. Let $\{H_i | 1 \leq i \leq n\}$ and $\{H'_i | 1 \leq i \leq m\}$ be the sets of head constraints of H and H' respectively, then the tuple*

$$\left(\begin{array}{c} g \wedge g' \wedge H_{i_1} = H'_{j_1} \wedge \ldots \wedge H_{i_k} = H'_{j_k} \\ (B, H_k \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m}) \ =\!\downarrow\!= \ (B', H'_k \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n}) \end{array} \right)$$

is called a critical pair *of the two rules R and R'. $\{i_1, \ldots, i_n\}$ and $\{j_1, \ldots, j_m\}$ are permutations of the sets $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ respectively and $k \leq \min(m, n)$. In addition, either $\exists H_{i_p} : p \in \{i_1, \ldots, i_k\} \wedge H_{i_p} \in H_r$ or $\exists H'_{j_p} : p \in \{j_1, \ldots, j_k\} \wedge H'_{j_p} \in H'_r$. (The symbol $=\!\downarrow\!=$ is merely used as a syntactical separator.)*

We can now consider the issue of joinability at the level of critical pairs.

[1] Prolog also contains non-monotonous constraints, e.g. `var(X)` may hold initially, but no longer when for example `X = a` is added.

Definition 4.7 *A critical pair* $(G,(B,H) =\downarrow= (B',H'))$ *is called* joinable *if the execution states* $state(G,B,H)$ *and* $state(G,B',H')$ *are joinable, where*

$$\begin{array}{rcl} state(G,B,H) & = & \langle B, num(H), G, history(\#H-1)\rangle_{\#H} \\ num([H_0,\ldots,H_n]) & = & [H_0\#0,\ldots,H_n\#n] \\ history(i) & = & \left\{ I +\!\!+ [r] \;\middle|\; \begin{array}{l} (r@H <=> g|C) \in \mathcal{P} \wedge \\ I \subseteq \{0,\ldots,i\} \wedge \#I = \#H \end{array} \right\} \end{array}$$

The following theorem is given in (Abdennadher 1997) regarding the joinability of critical pairs.

Theorem 4.3 *A CHR program* $\mathcal{P}$ *is locally confluent if and only if all its critical pairs are joinable.*

To extend this theorem from local confluence to full confluence, we need to introduce the notion of terminating CHR programs.

Definition 4.8 (Terminating Program) *A CHR program* $\mathcal{P}$ *is terminating, if all derivations are finite.*

The main confluence result for CHR is then:

Corollary 4.1 *A terminating CHR program* $\mathcal{P}$ *is confluent if and only if all of its critical pairs are joinable.*

It is possible to build a confluence checker for terminating CHR programs from the above theorems and definitions. The checker should compute all critical pairs and verify whether they are joinable by computing final states from them. If the final states are variants of each other, they are joinable. Otherwise they are non-joinable and the CHR program is non-confluent.

4.4.1 Related Notions

Canonical Programs *Canonicity* is a stronger property for CHR programs than confluence. Canonicity provides a link between the operational behavior of the program and its underlying constraint theory. Initial states should not only produce syntactically identical final states if they are syntactically identical, but also if they are logically equivalent.

Definition 4.9 (Canonical Program) *A confluent CHR program* $\mathcal{P}$ *is* canonical *if:*

$$\begin{array}{c} \forall \sigma_1,\sigma_2,\sigma_1',\sigma_2' : (\sigma_1 \rightarrowtail^* \sigma_1') \wedge (\sigma_2 \rightarrowtail^* \sigma_2') \wedge \\ ([\![\mathcal{P}]\!], \mathcal{D}_b \models meaning(\sigma_1) \Leftrightarrow meaning(\sigma_2)) \\ \Downarrow \\ \sigma_1' \text{ and } \sigma_2' \text{ are joinable} \end{array}$$

In (Stuckey and Sulzmann 2005) a sufficient, but not necessary, condition for canonical programs is given: confluent range-restricted[2] rules where all simplification rules are single-headed, form canonical programs. In general, however, showing that a program is canonical is as undecidable as showing that it is confluent: both require showing that all derivations are finite, which is undecidable (cfr. the halting problem).

Canonical CHR programs will be of particular interest in Chapter 9 where we study automatic implication checking for constraint solvers written in CHR.

Completion Completion is a technique to turn a non-confluent CHR program $\mathcal{P}$ into a confluent CHR program $\mathcal{P}'$ by adding new rules. The procedure is based on the well-known Knuth-Bendix completion (Knuth and Bendix 1970) for term rewriting systems (see Section 2.5.2). The added rules orient the critical pairs of the initial program: the CHR constraints in one of the final states of the critical pair are simplified to the CHR and built-in constraints of the other final state. Also, the built-in constraints of the one final state are added if the CHR and built-in constraints of the other final state are present. The main work on completion for CHR is (Abdennadher and Frühwirth 1998).

Refined Operational Semantics Confluence is defined in terms of the theoretical operational semantics ω_t. However, in (Duck, Stuckey, García de la Banda, and Holzbaur 2004) the following corollary is established.

Corollary 4.2 *A CHR program that is confluent with respect to ω_t, is also confluent with respect to ω_r.*

As the refined operational semantics ω_r is a particular instance of the theoretical operational semantics ω_t, clearly confluence with respect to ω_t also implies confluence with respect to ω_r. However, the other way around, confluence with respect to ω_r does not necessarily imply confluence with respect to ω_t.

Hence the discussed notion of confluence may not be very convenient for detecting problems in programs that were written with the refined operational semantics in mind. For example, particular critical pairs may involve one or more execution states that are unreachable under ω_r semantics. If these critical pairs are not joinable, non-confluence may be concluded unnecessarily for ω_r semantics.

In (Duck, Stuckey, García de la Banda, and Holzbaur 2004) a stronger partial confluence check is presented for ω_r. The check is partial in that it does not cope with built-in constraints.[3]

[2] A rule is *range-restricted* if any grounding of the head of the rule is also a grounding of the body.

[3] More specifically, it only deals with CHR programs for which the **Solve** transition of ω_r is never applicable.

4.5 Confluence Case Study: Union-Find

The confluence of the union-find implementations of Section 3.2 was analyzed with a small confluence checker written in Prolog and CHR. For the implementation of the naive union-find algorithm of Listing 3.3, eight non-joinable critical pairs were found. Two non-joinable critical pairs stem from overlapping the rules for `find`. Four non-joinable critical pairs stem from overlapping the rules for `link`. The remaining two critical pairs are overlaps between `find` and `link`.

We found one non-joinable critical pair that is unavoidable (and inherent in the union-find algorithm), three critical pairs that feature incompatible tree constraints (that cannot occur when executing allowed queries), and four critical pairs that feature pending link constraints (that cannot occur for allowed queries in the standard left-to-right execution order of ω_r). The critical pairs are discussed in more detail below. In the technical report (Schrijvers and Frühwirth 2004) associated with this work, we also add rules, both by hand and by automatic completion, to make the critical pairs joinable.

4.5.1 Inherent Non-Confluence

The non-joinable critical pair between the rule `findRoot` and `link` exhibits that the relative order of `find` and `link` operations matters. Recall that the two rules are:

```
findRoot @ root(X) \ find(X,A) <=> A=X.

link     @ link(C,B), root(C), root(B) <=> B ~> C, root(C).
```

The full critical pair is:

$$\left(\begin{array}{c} \texttt{true} \wedge \texttt{true} \wedge \texttt{root(X) = root(B)}, \\ (\texttt{A = X}, \texttt{root(X)} \wedge \texttt{link(C,B)} \wedge \texttt{root(C)}) \\ =\downarrow= \\ (\texttt{B ~> C} \wedge \texttt{root(C)}, \texttt{find(A,X)}) \end{array}\right)$$

From now on we use the notation in the table below for briefness. It lists in the first row the goal of the minimal execution state of the critical pair. To overlapping rules apply to this goal. The first rule is the `findRoot` goal and the second is the `link` rule. The constraints in the constraint stores of their respective final states are listed in the second and the third row of the table.

Overlap	`find(B,A),root(B),root(C),link(C,B)`
`findRoot`	`root(C),B~>C,A=B`
`link`	`root(C),B~>C,A=C`

It is not surprising that a `find` after a `link` operation has a different outcome if linking updated the root. As noted in Section 4.3, this update is unavoidable and inherent in the union-find algorithm.

4.5.2 Incompatible Tree Constraints Cannot Occur

The two non-joinable critical pairs for `find` correspond to queries where a `find` operation is confronted with two tree constraints it could apply to. Also the non-joinable critical pair involving the rule `linkEq` features incompatible tree constraints.

Overlap	A~>B,A~>D,find(A,C)
findNode	A~>B,A~>D,find(B,C)
findNode	A~>B,A~>D,find(D,C)
Overlap	root(A),A~>B,find(A,C)
findNode	root(A),A~>B,find(B,C)
findRoot	root(A),A~>B,A=C
Overlap	root(A),root(A),link(A,A)
linkEq	root(A),root(A)
link	root(A),A~>A

The conjunctions (`A~>B, A~>D`), (`root(A), A~>B`), (`root(A), A~>A`) and (`root(A), root(A)`) that can be found in the overlaps (and non-joinable critical pairs) correspond to the cases that violate the definition of a tree: a node with two parents, a root with a parent, a root node that is its own parent and a tree with two identical roots respectively.

We show that the overlapping conjunctions of the three non-joinable pairs cannot occur as the result of running the program for an allowed query. Observe that the rule `make` is the only one that produces a `root`, and the rule `link` is the only one that produces a `~>`. The rule `link` needs `root(A)` and `root(B)` to produce `A ~> B`, and it will absorb `root(A)`.

In order to produce one of the first three conjunctions, the link operation(s) need duplicate `root` constraints to start from. But only a query containing multiple copies of `make` (e.g. `make(A),make(A)`) can produce such duplicate `root` constraints. Since duplicate `make` operations are not allowed in queries, we cannot produce any of the non-joinable critical pairs.

4.5.3 Pending Links Cannot Occur

The remaining four non-joinable critical pairs stem from overlapping the rule for `link` with itself. They correspond to queries where two `link` operations have at least one node in common such that when one link is performed, at least one node in the other link operation is not a root anymore. When we analyze these non-joinable critical pairs we see that the two conjunctions (`A~>C,link(A,B)`) and (`A~>C,link(B,A)`) are dangerous.

Overlap	root(A),root(B),link(B,A),link(A,B)
link	root(B),A~>B,link(A,B)
link	root(A),link(B,A),B~>A
Overlap	root(A),root(B),root(C),link(B,A),link(C,B)
link	root(C),A~>B,B~>C
link	root(A),root(C),link(B,A),B~>C
Overlap	root(A),root(B),root(C),link(B,A),link(A,C)
link	root(B),root(C),A~>B,link(A,C)
link	root(B),C~>A,A~>B
Overlap	root(A),root(B),root(C),link(B,A),link(C,A)
link	root(B),root(C),A~>B,link(C,A)
link	root(B),root(C),link(B,A),A~>C

Once again, we argue that the critical pairs cannot arise in an allowed query. `link` is an internal operation, it can only be the result of a `union`, which is an external operation. In the union, the link constraint gets its arguments from `find`. In the standard left-to-right execution order of most sequential CHR implementations (Duck, Stuckey, García de la Banda, and Holzbaur 2004), first the two find constraints will be executed and when they have finished, the link constraint will be processed. In addition, no other operations will be performed in between these operations. Then the results from the find constraints will still be roots when the link constraint receives them. Note that such an execution order is always possible, provided `make` has been performed for the nodes that are subject to union (as is required for allowed queries).

4.5.4 Conclusion

In this section we have studied the confluence of the naive union-find implementation. The study has confirmed the expected inherent destructive update and hence non-confluence of the union-find algorithm: a `find` before or after a `union` operation may yield a different result.

Moreover the study has revealed some problems with the usability of the confluence checker: it returns too many irrelevant non-joinable critical pairs. We have found two main causes for these irrelevant non-joinable critical pairs:

- The confluence checker does not restrict itself to the allowed queries.
- The confluence checker not does restrict itself to the refined operational semantics, for which the program was written.

In both these cases the confluence checker considers execution states that are not reachable during intended uses of the program.

A similar study for the optimal union-find implementation (Schrijvers and Frühwirth 2004) yields similar results. In addition to the above two causes of irrelevant non-joinable critical pairs, a third cause was also identified:

- The confluence checker does not consider a more semantical criterion to compare final states.

For the optimal union-find algorithm, some non-joinable critical pairs may result in different trees containing the same elements. On the level of the algorithm, these different trees are semantically equivalent and hence their syntactical difference is irrelevant.

The study of the optimal program also gives some insight in the scalability of the method. While the optimal program is only slightly larger than the naive one, the checker finds 73 different non-joinable critical pairs, an order of magnitude difference. Clearly it may quickly become unmanageable to sort out the non-joinable critical pairs in relevant and irrelevant ones. Hence, it is necessary to come up with more refined techniques that remedy the above usability issues and that are relevant for practical CHR programs.

4.6 Time Complexity

The main general complexity results of CHR are (Frühwirth 2002a; Frühwirth 2002b). The complexity measure is basically the derivation length times the sum of the costs of rule trials and a rule application. The dominating cost factor is the computation of the cross product of all constraints in a goal when trying to apply a rule. Often, however, this cost can be made constant by relying on execution orders (scheduling) of constraints and indexing techniques.

More precisely, in (Frühwirth 2002b) the following worst-case time complexity bound is derived for the runtime $T(Q)$ of executing a query Q with n atomic CHR constraints. The result assumes that the CHR program consists of simplification rules only and it is independent of any concrete CHR implementation, but rather addresses the inherent complexity of CHR.

$$T(Q) = \mathcal{O}(D \sum_i ((c+D)^{n_i}(O_{H_i} + O_{G_i}) + (O_{C_i} + O_{B_i}))),$$

where

c	is the number of constraints in the query Q
$D(= \|Q\|)$	is the worst case derivation length of the query Q
i	ranges over the rules in the CHR program
n_i	is the number of heads in rule i

O_{H_i} is the complexity of the head matchings for rule i

O_{G_i} is the complexity of the guard for rule i

O_{C_i} is the complexity of adding the right-hand side built-in constraints for rule i

O_{B_i} is the complexity of removing the left-hand side CHR constraints and of adding the right-hand side CHR constraints for rule i

No work has been published concerning the time complexity of CHR programs containing propagation rules. It is an important open problem.

4.7 Time Complexity Case Study: Union-Find

Based on the above general complexity results we will now study the time complexity of the optimal union-find implementation of Section 3.2.

These assumptions will underly our complexity analysis:

- The query consists of N `make/1` constraints for different elements (nodes), followed by M `union/2` and `find/2` constraints, arbitrarily ordered. Hence, $c = \mathcal{O}(N + M)$.
- All of $O_{H_i}, O_{G_i}, O_{C_i}$ and O_{B_i} are constants. This means that a single successful rule application can be done in constant time, whereas the total cost of multiple attempted rule applications may still be above constant time.
- Simpagation rules are rewritten into the corresponding simplification rules.

4.7.1 Worst-Case Time Complexity

We derive the derivation length from a termination norm that uses the following mapping of constraints and constants onto natural numbers:

- $|C \wedge D| = |C| + |D|$.
- $|$`A=B`$| = |$`root(A,N)`$| = |$`A~>B`$| = 0$.
- $|$`make(A)`$| = |$`link(A,B)`$| = 1$.
- $|$`find(A,B)`$| = |$`A`$| + 1$.
- $|$`union(A,B)`$| = |$`A`$| + |$`B`$| + 4$.
- $|$`A`$|$ is the path length (distance) from the node `A` to the root of the tree.

That is, the termination norm and the derivation length of a query depends only on the depth of the recursion of `find/2`. The upper bounds for the depth of the tree - and thus the path length - are given by the number N of elements and also by the rank, the second argument of `root/2`. The order of the derivation length of a single find, i.e. $|\texttt{A}|$ of its first argument, can be bound by $|\texttt{A}| \leq log_2(N)$. This can be easily shown by induction on the rank.

Putting it all together, the derivation length of a query with N `make/1` constraints and M `union/2` and `find/2` constraints is $D = \mathcal{O}(N + Mlog(N))$ and also $c + D = \mathcal{O}(N + Mlog(N))$.

Taking into account that $\max_i(n_i) = 3$, we get the following complexity bound:

$$\mathcal{O}((N + Mlog(N))^4)$$

4.7.2 Optimal Time Complexity

The above complexity bound is not even close to the known complexity bound of the optimal imperative algorithm. Hence, we will abandon the general complexity results and instead we will establish the time complexity of our CHR programs by first showing that they are operationally equivalent to the respective imperative algorithms. By showing next that all the individual computation steps in the CHR program have the same complexity as their imperative counterparts, we then have effectively proven that the overall time complexity is identical to that of imperative implementations.

Operational Equivalence We start by considering the naive algorithm. Because of the refined operational semantics of CHR, the query of `make/1`, `union/2` and `find/2` constraints (and any other conjunction of constraints) is evaluated from left to right, just as is the case for equivalent calls for the imperative program.

Because of this execution order, the operation constraints behave just as their imperative counterparts. The imperative `if-then` and `if-then-else` constructs are encoded as multiple rules. The appropriate rule will be chosen because of a combination of different matchings, partner constraints and guards.

Moreover, the recursion depth for the `find/2` constraint is equal to the path from the initial node to the root just as in the imperative algorithm. The unification in the body of the `findRoot` rule does not wake up any constraints, since the variable that is bound to a constant does not occur in any other constraint processed so far.

It is clear from the CHR program and the refined operational semantics, that there is only ever at most one operation constraint in the constraint store. Moreover, whenever a data constraint is called, the operation constraint has already

been removed. Thus a data constraint will never trigger any rule, because of lack of the necessary partner constraint.

Time Complexity Equivalence Now that we have shown the operational equivalence of the CHR program with the imperative algorithm, we still need to show that the time complexities of the different computation steps (corresponding to rule applications) are also equal.

The following time complexity assumptions of a CHR implementation are reasonable. They are effectively implemented by the SICStus (Intelligent Systems Laboratory 2003), HAL (Holzbaur, García de la Banda, Stuckey, and Duck 2005) and the K.U.Leuven CHR system (see Chapter 6). All of the following operations of the refined operational semantics take constant time:

- The `Activate` transition, excluding the cost of adding the constraint to the constraint store.
- The `Drop` transition, i.e. ending the execution of a constraint.
- The `Default` transition, i.e. switching from trying one rule to trying the next rule.
- Matching for Herbrand variables and constants, given a bounded reference chain length. This occurs in the `Simplify` and `Propagate` transitions.
- Instantiating a variable that does not occur in any constraints, i.e. an obvious optimization of the `Solve` transition.
- Checking simple arithmetic built-in constraints like `>=` and `min`.

In addition, we make the following complexity assumptions for the union-find programs:

1. The cost of finding all constraints with a particular value in a particular argument position is constant, even if there are no such constraints. The cost of obtaining one by one all constraints from such a set is proportional to the size of the set.
2. The CHR constraint store allows constant time addition and deletion of any constraint.
3. If more than one partner constraint has to be found, an ordering of lookups is preferred, if possible, such that the next constraint to look up shares a variable with the previously found constraints and the active constraint.

The above complexity assumptions can be realized in practice by appropriate indexing, i.e. constraint store lookup based on shared variables. The last item

is a heuristic presented in (Holzbaur, García de la Banda, Stuckey, and Duck 2005) and implemented in the HAL and the K.U.Leuven CHR system. In Section 6.3.3 we will discuss appropriate constraint store data structures that satisfies the remaining assumptions.

From these assumptions it is clear that processing a data constraint takes constant time: the constraint is called, some rules are tried, some partner constraints that share a variable with the active constraint are looked for, but none are present, and finally the call ends with inserting the data constraint into the constraint store.

From these assumptions and the constant time calling of data constraints, it follows that all rule tries and applications with an active constraint take constant time. Hence our naive CHR implementation has the same time complexity properties as the naive imperative algorithm.

The proof of operational equivalence and equivalent complexity of the optimized algorithm and CHR program is similar.

Because of this equivalence with the imperative algorithm, our CHR program also has worst-case time complexity $\mathcal{O}(M + Nlog(N))$ and amortized time complexity $\mathcal{O}(M + N\alpha(N))$.

4.8 Conclusion

In this chapter we have given an overview of the most important theoretical properties of CHR programs: declarative semantics, confluence and time complexity.

The first two properties are of interest to programmers because they are indicators for the correctness of the program. The declarative semantics gives a logical meaning to a CHR program. Soundness of this logical meaning with respect to the intended logical theory may be verified by the programmer. Confluence guarantees the uniqueness of the final state for any initial state. Almost always, only a single final state is intended for any initial state and non-confluence is a clear indicator that the program does not live up to this intention.

The worst-case time complexity of a CHR program gives an upper bound on the computational cost in terms of the length of the query.

We have illustrated and studied these properties for the union-find programs of Section 3.2. The case studies show several shortcomings in the current state of verifying these properties. Namely, the current notion of confluence for all possible queries is too crude. Often only a particular set of queries are actually used for a particular CHR program. Moreover, the final state need not always be unique: particular aspects of the final state are of no relevance and a variation in them is hence permitted. Similarly, the formula for a time complexity bound considers all possible queries and not just the ones of interest. In addition, the formula is not geared towards the refined operational semantics and hence gives a much cruder bound than necessary for existing CHR systems.

Finally, we have shown that our CHR implementation of the union-find algorithm with path-compression and union-by-rank has the same optimal time complexity as its imperative counterpart, given certain time complexity assumptions for the CHR system.

The declarative semantics and the confluence properties of the union-find programs were published in the technical report (Schrijvers and Frühwirth 2004) and presented at the Workshop on (Constraint) Logic Programming (WCLP'05) (Schrijvers and Frühwirth 2005). The time complexity analysis of the programs based on the complexity formula was also published in the technical report. The optimal complexity bound was an important part of the *programming pearl* accepted for publication by the journal Theory and Practice of Logic Programming (Schrijvers and Frühwirth 2005).

Chapter 5

The Implementation of CHR: A Reconstruction

5.1 Introduction

There are quite a number of different CHR systems, compilers and interpreters. Each has its own approach towards execution and compilation of CHR. A short historical overview of these systems is given in Chapter 6.

In this chapter we focus on the compilation schema of only one CHR system: the CHR system developed by Christian Holzbaur in co-operation with Thom Frühwirth (Holzbaur and Frühwirth 1999). This is the CHR system included in SICStus Prolog (Intelligent Systems Laboratory 2003) and in Yap (Santos Costa, Damas, Reis, and Azevedo 2004). It is generally considered as the reference implementation of CHR and the refined operational semantics (Duck, Stuckey, García de la Banda, and Holzbaur 2004) were written to formalize its behavior.

We have studied the reference implementation using the limited available documentation of (Holzbaur and Frühwirth 1999; Holzbaur and Frühwirth 2000) and dissecting its generated output. The rest of this chapter comprises a description of the reference implementation's compilation schema and its data structures. In Section 5.2, we start from a simplified compilation schema that closely follows the refined operational semantics and one by one add all the optimizations included in the reference implementation. The optimizations are motivated in terms of the refined operational semantics with the help of our reconstructed reasonings.

Moreover, Section 5.4 presents two soundness proofs concerning the Prolog compilation schema with respect to the refined operational semantics. Finally, the content of this chapter is summarized in Section 5.5.

5.2 Compilation Schema

In this section we introduce a Prolog compilation schema for CHR: the schema is a generic blueprint for transforming a CHR program into executable Prolog code. In our presentation we will relate the different implementation aspects with the different components of the execution state $\langle A, S, B, T\rangle_n$ and the transition rules of the refined operational semantics.

First, in Section 5.2.1 we present a basic schema that was inspired by the refined operational semantics and the reference implementation (Holzbaur and Frühwirth 1999). It mainly covers the implementation of the transition rules and the execution stack A.

Then, in Section 5.2.2, we elaborate on the actual constraint representation, i.e. the implementation of numbered constraints and the (distributed) implementation of the propagation history T. The implementation of the constraint stores S and B is covered in Section 5.2.3

Optimizations to the basic schema and data structures as well as a compilation example are covered in Section 5.3.

5.2.1 Basic Compilation Schema

The basic compilation schema we present is simpler than the initial compilation schema of the SICStus reference implementation given in (Holzbaur and Frühwirth 1999). However, the relation of the basic schema to the refined operational is more obvious. This makes it easier to justify optimizations of the schema by reasoning about the refined operational semantics. Formal proofs of some optimizations will be given in Section 5.4.

The compilation schema maps the execution stack A of the refined operational semantics onto the implicit execution stack of the host language Prolog. A sequence of goals is pushed onto the front of the execution stack by simply calling the conjunction of the goals.

Listing 5.1 gives the basic compilation schema for the high-level control flow of an active constraint `c/m`. The first predicate, `c/m`, corresponds with the **Activate** transition: the new constraint is inserted into the constraint store, assigned a unique identifier (`ID`) and put on the execution stack.

The second predicate, `c_occurrences/(m+1)`, takes care of trying the o different occurrences of constraint `c`; it corresponds with the succession of **Default** transitions and the final **Drop** transition. This predicate is also called directly for the **ReActivate** transition.

In addition to the above control flow skeleton, there is of course also the code for the individual occurrences. We will consider this code for CHR rules of the following general form:

$\mathtt{r}_j$ `@` $\mathtt{c}_{r+1}(\mathtt{X}_{r+1,1},\ldots,\mathtt{X}_{r+1,m_{r+1}}),\ldots,\ \mathtt{c}_n(\mathtt{X}_{n,1},\ldots,\mathtt{X}_{n,m_n})$ `\`

```
c(X1,...,Xm) :-
        insert_in_store_c(X1,...,Xm,ID),
        c_occurrences(X1,...,Xm,ID).

c_occurrences(X1,...,Xm,ID) :-
        c_occurrence1(X1,...,Xm,ID),
        ⋮
        c_occurrenceo(X1,...,Xm,ID).
```

Listing 5.1: The Compilation Schema for Succession of Multiple Occurrences

$$\mathtt{c}_1(\mathtt{X}_{1,1},\ldots,\mathtt{X}_{1,m_1}),\ldots,\mathtt{c}_r(\mathtt{X}_{r,1},\ldots,\mathtt{X}_{r,m_r})$$
$$\texttt{<=> Guard | Body.}$$

Assume that $\mathtt{c}_l/\mathtt{m}_l = \mathtt{c/m}$ and that $\mathtt{c}_l\mathtt{m}_l$ in rule r_j is the ith occurrence of `c/m`. Then the code given in Listing 5.2 corresponds with a **Simplify** or **Propagate** transition for occurrence i in rule $\mathtt{r}_j$, depending on whether $l \leq r$ or $l > r$.

The code iterates over all possible partner constraints: the predicate $\mathtt{universal\_lookup\_c}_k$ produces an iterator[1] $\mathtt{Iter}_k$ over $\mathtt{c}_k$ constraints and the nested calls to $\mathtt{c\_occurrence}_i\mathtt{\_k}$ gather all the necessary candidate partner constraints. Finally in the most deeply nested call, to $\mathtt{c\_occurrence}_i\mathtt{\_n}$, it is verified whether all the involved constraints are still active (i.e. present in the constraint store), whether they are all mutually different, whether the guard succeeds and whether no tuple is already present in the propagation history for this combination of partner constraints. If all the tests succeed, the transition can be applied: a new tuple is added to the propagation history, the body is executed. Then execution continues with another possible combination of partner constraints.

Example 5.1 The compiled code for the `gcd` program of Example 2.1 derived from the basic schema is:

```
1  gcd(I) :-
2          insert_in_store_gcd(I,ID),
3          gcd_occurrences(I,ID).
4
5  gcd_occurrences(I,ID) :-
6          gcd_occurrence1(I,ID),
7          gcd_occurrence2(I,ID),
8          gcd_occurrence3(I,ID).
9
10 % gcd(0) <=> true.
```

[1] See Section 5.2.3 for details on the implementation of `universal_lookup_c`.

```
1  c_occurrence_i(X_1,...,X_m,ID) :-
2        universal_lookup_c_1(Iter_c_1),
3        c_occurrence_i_2(Iter_c_1,X_1,...,X_m,ID).
4
5  c_occurrence_i_2(Iter_c_1,X_1,...,X_m,ID) :-
6        empty(Iter_c_1), !.
7  c_occurrence_i_2(Iter_c_1,X_1,...,X_m,ID) :-
8        next_c_1(Iter_c_1,X_{1,1},...,X_{1,m_1},ID_1,Rest_1),
9        universal_lookup_c_2(Iter_c_2),
10       c_occurrence_i_3(Iter_c_2,X_{1,1},...,X_{1,m_1},ID_1,Rest_1,X_1,...,X_m,ID).
11 ⋮
12 c_occurrence_i_n(Iter_c_n,Args) :-
13       empty(Iter_c_n), !,
14       c_occurrence_i_n-1(Args).
15 c_occurrence_i_n(Iter_c_n,Args) :-
16        next_c_n(Iter_c_n,X_{n,1},...,X_{n,m_n},ID_n,Rest_n),
17       ( alive(ID_1),
18         ⋮
19         alive(ID_n),
20         ID_1 \== ID_2,
21         ⋮,
22         ID_{n-1} \== ID_n,
23         Guard,
24         T = [r,ID_1,...,ID_n],
25         not_in_history(T)
26         ->
27               add_to_history(T),
28               kill(ID_1),
29               ⋮
30               kill(ID_r),
31               Body
32       ;
33               true
34       ),
35       c_occurrence_i_n(Rest_n,Args).
```

Listing 5.2: The Compilation Schema for a Kept Occurrence

```
11  gcd_occurrence1(I,ID) :-
12          ( alive(ID),
13            I == 0,
14            T = [1,ID],
15            not_in_history(T)
16          ->
17                  add_to_history(T),
18                  kill(ID),
19                  true
20          ;
21                  true
22          ).
23
24  % gcd(J) \ gcd(I) <=> J >= I | K is J - I, gcd(K).
25  gcd_occurrence2(I,ID) :-
26          universal_lookup_gcd(Iter),
27          gcd_occurrence2_2(Iter,I,ID).
28
29  gcd_occurrence2_2(Iter,I,ID) :-
30          empty(Iter), !.
31  gcd_occurrence2_2(Iter,I,ID) :-
32          next_gcd(Iter,J,ID2,Rest),
33          ( alive(ID2),
34            alive(ID),
35            ID2 \== ID,
36            J >= I,
37            T = [2,ID2,ID],
38            not_in_history(T)
39          ->
40                  add_to_history(T),
41                  kill(ID),
42                  K is J - I,
43                  gcd(K)
44          ;
45                  true
46          ),
47          gcd_occurrence2_2(Rest,I,ID).
48
49
50  % gcd(J) \ gcd(I) <=> J >= I | K is J - I, gcd(K).
51  gcd_occurrence3(J,ID) :-
52          universal_lookup_gcd(Iter),
```

```
53          gcd_occurrence3_2(Iter,J,ID).
54
55  gcd_occurrence3_2(Iter,J,ID) :-
56          empty(Iter), !.
57  gcd_occurrence3_2(Iter,J,ID) :-
58          next_gcd(Iter,I,ID2,Rest),
59          ( alive(ID),
60            alive(ID2),
61            ID \== ID2,
62            J >= I,
63            T = [2,ID,ID2],
64            not_in_history(T)
65          ->
66                  add_to_history(T),
67                  kill(ID2),
68                  K is J - I,
69                  gcd(K)
70          ;
71                  true
72          ),
73          gcd_occurrence3_2(Rest,I,ID).
```

5.2.2 Constraint Representation

Upto now we have used the generic `ID` as the identifier of a numbered constraint `c(`X_1`,...,`X_m`)#i`. In practice, a bit more information needs to be associated with a numbered constraint than just a unique identifier. In the CHR reference implementation (Holzbaur and Frühwirth 1999) in SICStus the following representation is used for a numbered constraint:

`suspension(ID,MState,Continuation,MGeneration,MHistory,c,`X_1`,...,`X_n`)`

The representation is a term with functor `suspension` and a number of fields (or arguments). This term representation is called *constraint suspension* or *suspension* for short.

The meaning of the fields is listed in Table 5.1.

Some of the fields in the term are mutable; this is indicated with the initial capital `M` in their name. They are implemented using the non-standard Prolog built-in `setarg/3` that destructively updates an argument of a term.

`ID`	The unique constraint identifier. In practice it is an integer.
`MState`	The state of the suspension. It takes one of three values:
`not_stored`	The constraint has not been stored yet. This is explained along with the late storage optimizations in Section 5.3.
`stored`	The constraint has been stored in the CHR constraint store.
`removed`	The constraint has been removed from the CHR constraint store.
`Continuation`	The continuation goal to be executed during a **ReActivate** transition. This calls the code for the first occurrence.
`MGeneration`	The generation number discussed later in Section 5.3 along with the generation optimization.
`MHistory`	Part of the propagation history.
`c`	The constraint functor.
$\texttt{X}_1,\ldots,\texttt{X}_n$	The arguments of the constraint.

Table 5.1: Meaning of the constraint suspension fields

5.2.3 Constraint Stores and Built-in Constraints

Built-in Constraint Store We call a constraint that can be imposed a *tell constraint*. In plain Prolog the sole tell constraint is the unification constraint with corresponding Herbrand constraint theory $\mathcal{H}$ (see Section 2.3.1). Hence, for a CHR system with plain Prolog as a host language, unifications are the only built-in constraints that cause **Solve** transitions. Other constraints are supported by Prolog that can only be used for implication checking, i.e. whether they are implied by the built-in constraint store. We call such a constraint that can be checked for implication an *ask constraint*. An example is `ground(T)` which checks whether term `T` is ground with respect to the current built-in constraint store. Ask constraints can be used in the guard of a CHR rule.

The implementation of the built-in unification constraint store is of course left to Prolog. Most Prolog systems implement unification as the naive union-find algorithm (see Section 3.2) extended with support for terms; this is the way it is conceived in the WAM, the widely implemented abstract instruction set for Prolog (Warren 1983; Aït-Kaci 1991).

As said before, the point where Prolog's unification constraints need to interact with CHR is in the **Solve** transition. The addition to the built-in constraint store is of course already handled by Prolog, but the **Solve** transition also needs to add constraints that are in the CHR constraint store to the execution stack. The **Solve** transition leaves some choice in what constraints are actually to add to the execution stack. For the reference implementation (Holzbaur and Frühwirth 1999) it was chosen to only add constraints to the execution stack which are *affected* by the unification.

Definition 5.1 (Affected) *We say that a CHR constraint c is affected by a unification constraint $x = y$ with respect to built-in constraint store B and CHR constraint store S if:*

$$
\begin{array}{lll}
\exists a, b : & & \\
 & \mathcal{H} \not\models B \rightarrow a = b & \\
\wedge & \mathcal{H} \models B \wedge x = y \rightarrow a = b & \\
\wedge & \mathcal{H} \models B \rightarrow a \in \mathit{term\_vars}(c) & \\
\wedge & \quad \mathcal{H} \models B \rightarrow \mathit{nonvar}(b) & \\
 & \vee \quad \exists d \in S : b \in \mathit{term\_vars}(d) &
\end{array}
$$

Summarized we say that a CHR constraint is affected by a unification, if a variable occurs in the CHR constraint that is unified with a nonvariable term or with another variable in a CHR constraint.

The choice to only push affected CHR constraints onto the execution stack is a sensible one, because the affected property provides a clear link with the

unification.

In Section 5.4.2 we will study the soundness of the pushing of affected CHR constraints with respect to the refined operational semantics.

Attributed variables (Holzbaur 1992) is an extension to Prolog which is useful in detecting when a variable is affected. This feature allows for marking variables, turning them into attributed variables. The unification of an attributed variable with another attributed variable or nonvariable is intercepted in order to run custom code. In addition to simply marking variables for the interception of unification, also additional updateable data can be associated with the variables.

We will use the interface described in (Demoen 2002) throughout this text. While it is not exactly the same as that of SICStus Prolog used in (Holzbaur and Frühwirth 1999), the differences are not relevant for our discussion. The following predicates make up the attributed variables interface:

`put_attr(`*Var*`,`*Mod*`,`*Attr*`)`
: Associate data *Attr* with variable *Var* for module *Mod*. Replaces previously associated data for that module. Turns the variable into an attributed variable.

`get_attr(`*Var*`,`*Mod*`,`*Attr*`)`
: Retrieve the associated data *Attr* from attributed variable *Var* for module *Mod*. Fails if there is no associated data for the module.

`del_attr(`*Var*`,`*Mod*`)`
: Dissociate any associated data of variable *Var* for module *Mod*.

`attvar(`*Var*`)`
: Succeed if the variable *Var* is an attributed variable. Fail otherwise.

Mod`:attr_unify_hook(`*Attr*`,`*Term*`)`
: The predicate defining the code to be executed after a unification of a variable with associated data *Attr* for module *Mod* to a term *Term*. This predicate is sometimes called the (interrupt) handler.

The **Solve** transition can now be realized as follows. Every variable that appears in a CHR constraint is turned into an attributed variable and as data all the suspensions of the CHR constraints it appears in are associated with it. When the attributed variables is unified with a non-variable or other attributed variables, the custom code that is run is code that puts all the associated constraint suspensions on the execution stack. Such an associated suspension is put on the execution stack by simply calling its continuation goal (see Section 5.2.2).

Guards The guard of a CHR rule consists of built-in constraints in the host language. In Prolog these built-in constraints are primarily term equality constraints.

Besides the explicit guard, some implicit built-in constraints, called *matching*, are also part of the guard.

Every rule is first turned into an equivalent rule in Head Normal From (see Section 2.4.1) for the purpose of obtaining the guard code that is inserted into the compilation schema.

CHR Constraint Store The CHR constraint store serves two purposes. The first is to collect identified CHR constraints for the purpose of re-activation during a **Solve** transition, has already been covered above. The second purpose is to be able to look up identified CHR constraints that may serve as partner constraints to the active constraint in the matching of a **Simplify** or **Propagate** transition. This corresponds with the `universal_lookup_c` and the `existential_lookup_c` (introduced in Section 5.3.1) functionality mentioned in the compilation schema.

In the reference implementation (Holzbaur and Frühwirth 1999), the CHR constraint store is implemented with the help of a global backtrackable variable that we refer to as `chr_store`. In this text we will use the interface implemented in hProlog and SWI-Prolog:

`b_getval(`$Name$`:,`$Value`)`
: Return the value $Value$ of the global variable with name $Name$:.

`b_setval(`$Name$`:,`$Value`)`
: Set the value of the global variable with name $Name$: to $Value$.

The value of `chr_store` is a term `v(`$List_1$`,...,`$List_n`)` where $List_i$ is the list of suspensions of the constraint c_i currently in the store. Initially all the lists are empty. Whenever a new constraint suspension is inserted into the constraint store, it is added to the appropriate list and whenever a constraint is removed from the constraint store, it is deleted from the appropriate list.

The iterator returned by the universal lookup is simply the appropriate list of `chr_store`. The predicates `empty/1` and `next_c`$_i$`/`$(m_i + 3)$ are realized through appropriate matching. Similarly, the existential lookup is realized by applying the well-known `member/2` predicate to the appropriate list, which generates all the suspensions in the list through backtracking.

In addition to the above global term of lists, also (potentially) much faster *indexes* are maintained in (Holzbaur and Frühwirth 1999). Every variable X that appears in the argument of a constraint is made an attributed variable with associated attribute data a similar term `v(`$List_{1,X}$`,...,`$List_{n,X}`)`. However, now $List_{i,X}$ contains only the suspensions of constraints in which X appears.

Whenever such a variable X is known at the time of a universal or existential lookup this variable's attribute data is used instead of the data contained in `chr_store`. Potentially this list is much smaller and it may reduce the number of partner constraints tried in vain tremendously.

Example 5.2 illustrates the use of an index for the lookup of a partner constraint.

Example 5.2 Consider the partner lookup code for the head `c1(X),c2(X)` with active occurrence `c1` is:

```
c1_occurrence1(X1,ID1) :-
      universal_lookup_c2_via(X1,Iter_c2),
      c1_occurrence1_2(Iter_c2,X1,ID1).
```

where `universal_lookup_c2_via/2` is roughly implemented as:

```
universal_lookup_c2_via(X,Iter) :-
        ( term_variables(X,[V|_]) ->
                get_attr(V,mod,v(_,Iter))
        ;
                universal_lookup_c2(Iter)
        ).
```

5.3 Simple Optimizations

In this section we list the simple optimizations that have been applied to the basic compilation schema in (Holzbaur and Frühwirth 1999).

In (Holzbaur and Frühwirth 1999) many optimizations have been applied to the basic compilation schema of the previous section. Here we list them and discuss the most straightforward ones. The more elaborate ones are discussed in extenso in other parts of this text.

We have grouped these optimizations into three categories: *semantical* optimizations, *host language* optimizations and *data structure* optimization. Semantical optimizations, covered in Section 5.3.1, are based on reasoning about the operational semantics of CHR. Host language optimizations, covered in Section 5.3.2, are based on reasoning about the efficiency of code in the host language. Data structure optimizations, covered in Section 5.3.3, deploy more efficient data structures.

Finally, we illustrate the presented schemas and optimizations by showing the compiled code for a CHR program in Section 5.3.4.

5.3.1 Semantical Optimizations

Propagation History Maintenance A rule application that removes a constraint can never be repeated with that constraint. So no propagation history needs

to be maintained for such a rule. In particular, simplification and simpagation rules remove constraints, while propagation rules do not. Hence, the propagation history needs to be maintained for propagation rules only.

Simplification Transition For a **Simplify** transition it is not necessary to iterate over all possible combinations of partner constraints. After the transition is applied for the first time, the active constraint is no longer active and no more applications are possible. Hence the universal lookup is replaced with an existential one in Listing 5.3. An existential lookup generates all candidate partner constraints through backtracking instead of providing an explicit iterator. This much simplifies the generated code.

```
1  c_occurrence_i(X_{i,1},...,X_{i,m_i},ID_i) :-
2          ( alive(ID_i),
3            existential_lookup_c_1(X_{1,1},...,X_{1,m_1},ID_1),
4            ⋮
5            existential_lookup_c_n(X_{n,1},...,X_{n,m_n},ID_n),
6            ID_1 \== ID_2,
7            ⋮,
8            ID_{n-1} \== ID_n,
9            Guard
10           ->
11                 kill(ID_1),
12                 ⋮
13                 kill(ID_r),
14                 Body
15         ;
16                 true.
17         ).
```

Listing 5.3: The Compilation Schema for a Removed Occurrence

Generations During the execution of the body in a **Propagate** transition, a built-in constraint may reactivate the constraint that was active in the **Propagate** transition. With respect to the refined operational semantics, it means that the same numbered constraint occurs more than once in the execution stack of an execution state. Section 5.4.1 contains a proof based on the refined operational semantics which states that in the above case, all but the topmost occurrence may be popped (i.e. removed) from the execution stack in CHR execution states.

Clearly the removal of numbered occurrences from the execution stack may considerably reduce the amount of transitions. For this reason, the situation is exploited in (Holzbaur and Frühwirth 1999). With every constraint identifier $\texttt{ID}_i$ a *generation number* $\texttt{G}_i$ is associated. This generation number is initialized to zero and incremented every time the constraint is reactivated. When the generation number of the active constraint is different before and after the execution of a rule body, the active constraint was reactivated during the execution of that body and may stop executing.

The generation optimization is most easily expressed as an optimization of the continuation-based formulation of the schema that is introduced in Section 5.3.2. The lines 1–7 of Listing 5.4 are replaced by the following:

```
⋮,
generation_number(ID_i,G_iold),
Body,
generation_number(ID_i,G_inew),
( alive(ID_i),
  G_iold == G_inew ->
        c_occurrence_i_n(Rest_n,...)
;
        true
)
```

The topmost constraint in the execution stack corresponds of course with the constraint with $\texttt{ID}_i$ in the above code. Popping this constraint corresponds with the `true` goal.

Late Storage The refined operational semantics state that a CHR constraint is inserted into the constraint store immediately in the **Activate** transition. However, often the lifetime of a CHR constraint is limited, i.e. it is removed from the constraint store not long after it has been activated, in particular by a **Simplify** transition where it is the active constraint.

The object of the *late storage* optimization is to defer constraint store insertion[2] as much as possible in order to avoid the actual insertion and removal operations, and in particular their associated overhead.

It is only sound to defer storage if the reachable final states with late storage are a subset of those with early storage. In (Holzbaur and Frühwirth 1999) a rather conservative approach is taken towards late storage. The process of storage is split into two parts: the creation of a constraint representation and the actual storage in the constraint store after that. The former is done immediately before the first **Propagate** occurrence whereas the latter is attempted just before the execution of the body of all **Propagate** transitions and after the last occurrence.

[2]Insertion is also known as storage.

While the actual storage may be attempted multiple times, it only updates the constraint store if the constraint is not present yet.

We will use the following predicates to refer to the separate operations:

`make_id_c`$_i$`(`$\mathtt{X}_{i,1}$`,...,`$\mathtt{X}_{i,m_i}$`,`$\mathtt{ID}_i$`)`
: Create a new constraint representation $\mathtt{ID}_i$.

`actually_insert_in_store_c`$_i$`(`$\mathtt{X}_{i,1}$`,...,`$\mathtt{X}_{i,m_i}$`,`$\mathtt{ID}_i$`)`
: Store the constraint if it has not already been stored.

As the code for the initial simplification occurrences is now used by both newly activated constraints without a representation (i.e. `ID` is a variable) and by reactivated constraints with a representation (i.e. `ID` is instantiated), the `kill` operations removing the constraint from the store are called conditionally as follows:

```
( var(ID) ->
        true
;
        kill(ID)
)
```

The conservative late storage approach of (Holzbaur and Frühwirth 1999) is extended and given a more formal treatment in Section 7.4.

5.3.2 Host Language Optimizations

Continuation-based Control Flow Listing 5.1 uses an explicitly sequential control flow whereas in the reference implementation (Holzbaur and Frühwirth 1999) a continuation-based approach is used. The corresponding predicate of Listing 5.1 becomes:

```
c_occurrences(X_{i,1},...,X_{i,m_i},ID_i) :-
        c_occurrence_1(X_{i,1},...,X_{i,m_i},ID_i).
```

This code simply tries the first occurrences. The first occurrence will take care of trying the next, if necessary.

Continuations can be split in two categories: the *live continuation* where the active constraint is alive and the *dead continuation* where the active constraint has been removed from the CHR constraint store. These categories will ensure that no **Simplify** or **Propagate** transition is attempted with a dead active constraint. Hence, the `alive(`$\mathtt{ID}_i$`)` is no longer needed as part of the tests before applying a transition. Instead these checks are moved to the points where the active constraint may just have been removed.

A dead continuation is simply the goal `true` corresponding with a **Drop** transition; nothing is left to be done with the active constraint. A live continuation tries the next possible rule application.

A **Simplify** transition always removes the active constraint. Hence after the execution of the body no other transitions with the active constriant needs to be attemped. When one of the tests for the transition fails, the live continuation proceeds by trying the next occurrence. This corresponds with replacing the `true` goal on line 16 in Listing 5.3 with:

$$\texttt{c\_occurrence}_{i+1}\texttt{(X}_{i,1}\texttt{,...,X}_{i,m_i}\texttt{,ID}_i\texttt{)}$$

A **Propagate** transition does not remove the active constraint directly. However, it may do so indirectly through the execution of the body of the CHR rule. Hence, the $\texttt{alive(ID}_i\texttt{)}$ test must be moved until after the execution of the body to decide between a live or a dead continuation. If the guard or any of the remaining `alive/1` tests on the partner constraints fail, only a live continuation is needed. Hence, the lines 31–35 of Listing 5.2 are replaced by those in Listing 5.4

```
1       ⋮,
2       Body,
3       ( alive(ID_i) ->
4               c_occurrence_i_n(Rest_n,...)
5       ;
6               true
7       )
8   ;
9       c_occurrence_i_n(Rest_n,...)
10  ).
```

Listing 5.4: Continuation Schema for the Propagate Transition

In addition to the above changes to the `alive/1` test for the active constraint, the **ReActivate** transition now also needs an $\texttt{alive(ID}_i\texttt{)}$ check. The reason is that the active constraint may have been removed from the constraint store since the execution of the **Solve** transition that has reactivated it. The following predicate may be used for the **ReActivate** transition:

```
c_reactivate(X_1,...,X_m,ID) :-
        ( alive(ID) ->
                c_occurrences(X_1,...,X_m,ID_i)
        ;
                true
        ).
```

Shallow Backtracking and Inlining In the reference implementation the above presented control flow through forward execution is replaced with *shallow backtracking* at some points. In the specialized code for **Simplify** transitions

the failure continuation is handled through shallow backtracking, i.e. instead of the if-then-else a cut is used to separate the *if* part from the *then* part and the *else* part is contained in a next clause of the same predicate. Moreover, the call to the predicate with the code for the next occurrence is *inlined*, i.e. the call is replaced with the corresponding body. The inlining avoids some overhead for the call and the shallow backtracking may reuse a choice-point for conditional parts of successive occurrences.

Listing 5.5 shows the compilation schema of a **Simplify** occurrence i and the next occurrence $i+1$ are compiled.

```
c_occurrence_{i,i+1}(X_{i,1},...,X_{i,m_i},ID_i) :-
        alive(ID_i),
        ⋮
        Guard
        !,
        ⋮
        Body.
c_occurrence_{i,i+1}(X_{i,1},...,X_{i,m_i},ID_i) :-
        ⋮
```

Listing 5.5: The Shallow Backtracking Compilation Schema

5.3.3 Data Structure Optimizations

Distributed Propagation History The implementation of the distributed propagation history deserves some further explanation. The propagation history is not maintained globally, but stored in a distributed fashion in the constraint suspensions. For every propagation transition that updates the propagation history, a tuple is added to the part of the propagation history maintained by the active constraint. Whenever the propagation history is to be checked for the presence of some tuple, it needs to be looked for only among the propagation histories of the involved constraints.

In the reference implementation in SICStus the data structure used for the distributed propagation history is an AVL tree in every constraint suspension. Although the worst case-lookup time in an AVL tree is $\mathcal{O}(\log n)$ whether the propagation history is distributed or not, on average the distributed approach should be cheaper.

Moreover, it is slightly more favorable for memory reuse: if all involved constraints become unreachable, their joint tuple becomes unreachable too and it can be reclaimed by a Prolog garbage collector. A global propagation history on the

other hand maintains the reachability of the tuple and custom memory management would be required to actually release the memory.

5.3.4 Optimized Compilation Example

Here we illustrate the above compilation schemas with optimizations on the `gcd` program presented in Example 2.1 on page 17 in Chapter 2.

Example 5.3

```
1  gcd(I) :-
2          gcd_occurrence_1_2_3(I,_).
3
4  % gcd(0) <=> true.
5  gcd_occurrence_1_2_3(I,ID) :-
6          I == 0,
7          !,
8          ( var(ID) ->
9                  true
10         ;
11                 kill(ID)
12         ).
13 % gcd(J) \ gcd(I) <=> J >= I | K is J - I, gcd(K).
14 gcd_occurrence_1_2_3(I,ID) :-
15         existential_lookup_gcd(J,ID2),
16         ID \== ID2,
17         J >= I,
18         !,
19         ( var(ID) ->
20                 true
21         ;
22                 kill(ID)
23         ),
24         kill(ID2),
25         K is J - I,
26         gcd(K).
27 % gcd(J) \ gcd(I) <=> J >= I | K is J - I, gcd(K).
28 gcd_occurrence_1_2_3(J,ID) :-
29         ( var(ID) ->
30                 make_id_gcd(J,ID)
31         ;
32                 true
33         ),
```

```
34          universal_lookup_gcd(Iter),
35          gcd_occurrence_3_2(Iter,J,ID).
36
37  gcd_occurrence_3_2([],J,ID) :-
38          gcd_drop(J,ID).
39  gcd_occurrence_3_2([ID2|Iter],J,ID) :-
40          ( ID2 = suspension(_,JState,_,_,_,_,I),
41            alive(JState),
42            ID2 \== ID,
43            J >= I ->
44                  kill(ID2),
45                    actually_insert_gcd(J,ID),
46                    generation_number(ID,GenOld),
47                    K is J - I,
48                    gcd(K),
49                    generation_number(ID,GenIew),
50                    ( alive(ID),
51                      GenIew == GenOld ->
52                          gcd_occurrence_3_2(Iter,J,ID)
53                    ;
54                            true
55                    )
56          ;
57                          gcd_occurrence_3_2(Iter,J,ID)
58          ).
59
60  gcd_drop(J,ID) :-
61          actually_insert_gcd(J,ID).
```

Compare the above compiled code to the unoptimized code of Example 5.1. Quite a number of optimizations have been applied:

- The *continutation-based control flow* has been introduced. The toplevel predicate `gcd/1` directly calls the code for the first occurrence at line 3, instead of calling an intermediate predicate that sequentially calls the code for all occurrences. Also, the `alive/1` tests for the active constraint at the lines 12, 34 and 59 in Example 5.1 have been dropped.
- The sequencing of the code for the different occurrences is realized through *shallow backtracking*: the toplevel clauses for each occurrence belongs to the same predicate `gcd_occurrence_1_2_3/2`.
- The code to *maintain the propagation history* (i.e. the lines 14–15 and 16 for the first occurrence, the lines 38–39 and 41 for the second occurrence and

the lines 63–64 and 66 for the third occurrence in Example 5.1) has been omitted altogether.

- The code for the second occurrence has been reduced to a single clause thanks to the specialization for *simplification transitions*: the replacement of the universal lookup with an existential one.
- Lines 49–55 implement the *generation optimization* for the third occurrence.
- The *late storage* optimization has been applied. This means that the early storage at line 2 of Example 5.1 has been removed. Instead, the constraint representation is created only at line 30 and the actual constraint store insertation occurs at line 45 just before the execution of the body of the third occurrence's rule. As a consequence, the removals of the active constraint in the first (lines 8–12) and the second (lines 19–23) occurrence are conditional. Moreover, an additional clause `gcd_drop/2` has been added to take care of inserting the active constraint after the last occurrence, if it has not yet been inserted.

5.4 Soundness Proofs of Optimizations

In this section we present two proofs that each establish the soundness of an aspect of the previously presented basic compilation schema with respect to the refined operational semantics.

First in Section 5.4.1 we establish the soundness of the generation optimization of Section 5.3.1. An earlier version of this proof was, to the best of our knowledge, the first soundness proof ever written for a CHR optimization. Second, in Section 5.4.2 we first illustrate an inconsistency in the **Solve** transition of the refined operational with respect to existing CHR implementations. Next we formulate an improved version of the **Solve** transition and then prove soundness of the CHR implementation with respect to it.

5.4.1 Soundness of the Generation Optimization

In this section we will prove the soundness of the generation optimisation explained in Section 5.3.1.

Paraphrasing Theorem 5.1, when an identified CHR constraint is pushed onto the execution stack while other occurrenced versions of the identified CHR constraint are already on the execution stack, it is allowed to remove the already present copies.

With soundness we mean that any final state reached through the optimization is also reachable without the optimization. Formally, the soundness theorem is the following:

Theorem 5.1 (Generation Optimization Soundness) *The generation optimization is sound with respect to the refined operational semantics:*

$$
\begin{array}{rl}
\forall \sigma, \sigma', \sigma_f, \sigma'_f, c, i, X, X', S, B, T, n, A_f, A'_f, S_f, S'_f, B_f, B'_f, T_f, T'_f, n_f, n'_f : & \\
\quad \sigma = \langle \{c\#i\} +\!\!+ X, S, B, T \rangle_n & \\
\wedge \quad \sigma' = \langle \{c\#i\} +\!\!+ X', S, B, T \rangle_n & \\
\wedge \quad \sigma'_f = \langle A'_f, S'_f, B'_f, T'_f \rangle_{n'_f} & \\
\wedge \quad \sigma'_f = \mathit{solve}(\sigma') & \\
\Longrightarrow \quad \sigma_f = \langle A_f, S_f, B_f, T_f \rangle_{n_f} = \mathit{solve}(\sigma) & \\
\wedge \quad S_f = S'_f & \\
\wedge \quad B_f = B'_f & \\
\wedge \quad T_f = T'_f & \\
\wedge \quad n_f = n'_f &
\end{array}
$$

where X' is identical to X with all occurrenced versions of $c\#i$ removed.

Proof: We will prove the soundness of a more restrictive case in Lemma 5.1: it is sound to remove the topmost occurrenced version of $c\#i$. By repeatedly applying this lemma to remove the first occurrenced $c\#i$ in X and by the transitivity of the soundness property, the theorem holds. □

In the theorem the state σ' corresponds with the one where the generation optimization is applied. The theorem does not explicitly distinguish between final success and failed states. Note though that σ_f and σ'_f have to be both either success or both failed states.

Lemma 5.1 *The more restrictive lemma states that it is allowed to remove the topmost occurrenced copy of an identified constraint from the execution stack when that identified CHR constraint is pushed onto the execution stack:*

$$
\begin{array}{rl}
\forall \sigma, \sigma', \sigma_f, \sigma'_f, c, i, X, Y, S, B, T, n, A_f, A'_f, S_f, S'_f, B_f, B'_f, T_f, T'_f, n_f, n'_f : & \\
\quad \sigma = \langle \{c\#i\} +\!\!+ X +\!\!+ [c\#i : j|Y], S, B, T \rangle_n & \\
\quad \wedge \quad \sigma' = \langle \{c\#i\} +\!\!+ X +\!\!+ Y, S, B, T \rangle_n & \\
\quad \wedge \quad \sigma'_f = \langle A'_f, S'_f, B'_f, T'_f \rangle_{n'_f} & \\
\quad \wedge \quad \sigma'_f = \mathit{solve}(\sigma') & \\
\quad \wedge \quad \forall X_1, X_2, k : X \neq X_1 +\!\!+ [c\#i : k|X_2] & \\
\Longrightarrow \quad \sigma_f = \langle A_f, S_f, B_f, T_f \rangle_{n_f} = \mathit{solve}(\sigma) & \\
\quad \wedge \quad S_f = S'_f & \\
\quad \wedge \quad B_f = B'_f & \\
\quad \wedge \quad T_f = T'_f & \\
\quad \wedge \quad n_f = n'_f &
\end{array}
$$

Proof:

By applying the semantical transition rules of CHR on σ, execution goes through states

$$\sigma = \sigma_0 \rightarrowtail \sigma_1 \rightarrowtail ... \rightarrowtail \sigma_l$$

with $\sigma_k = \langle X_k +\!\!+ [c\#i : j|Y], S_k, B_k, T_k \rangle_{n_k}$ for $k = 1..l$ such that σ_l is the first state where $X_k = \emptyset$ or where $\mathcal{D} \not\models B_k$, i.e. a failed state.

Alternatively, σ' goes with the same transitions through states

$$\sigma' = \sigma'_0 \rightarrowtail \sigma'_1 \rightarrowtail ... \rightarrowtail \sigma'_l$$

such that $\sigma_k = \langle X_k +\!\!+ Y, S_k, B_k, T_k \rangle_{n_k}$ for $k = 1..l$. This holds because transition rules depend on the topmost element of the execution stack only.

- Consider failure in σ_l, which is due to unsatisfiable constraints in B_l. Because σ'_l contains the same built-in constraint store B_l it is also a failure state. Hence, the lemma holds for this case with $\sigma_l = \sigma_f$ and $\sigma'_l = \sigma'_f$.
- If execution does not fail before $X_l = \emptyset$ in σ_l, Lemma 5.2 below shows that $\sigma_l \rightarrowtail \cdots \rightarrowtail \sigma_{l'}$ is the only possible transition sequence. From the common state $\sigma_{l'}$ on obviously the same final states are reachable. Hence, the current lemma also holds for this case.

□

Lemma 5.2 *Let σ_l, σ'_l and X_l be defined as in the proof of Lemma 5.1. Then the transition $\sigma_l \rightarrowtail \ldots \rightarrow \sigma'_l$ is the only possible transition sequence, if $X_l = \emptyset$.*

Proof: We make one more assumption about all X_k for $1 \leq k < l$, namely:

$$\nexists X_{k,t} : X_k = [c\#i|X_{k,t}]$$

- If this does not hold, then some σ_k with $k < l$:

  $$\sigma_k = \langle [c\#i|X_{k,t}] +\!\!+ [c\#i : j|Y], S_k, B_k, T_k \rangle_{n_k}$$

  For the maximum value of k now, assuming the current lemma holds with the additional assumption, it must be that σ'_l is the only state reachable from σ_l. Hence, if the current lemma holds under the additional assumption, it also holds without the additional assumption.

- The rest of this proof carries on with the additional assumption.

We will now show that the only possible sequence of transitions is of the form

$$\sigma_l \rightarrowtail \sigma_{l+1} \rightarrowtail ... \rightarrowtail \sigma_{l+o+1-j} \rightarrowtail \sigma'_l$$

such that $\sigma_{(l+k)} = \langle [c\#i : j + k|Y], S_l, B_l, T_l \rangle_{n_l}$ for $k = 0..(o + 1 - j)$, where o is the maximum occurrence number of c in program P.

Consider what transition rules apply in state σ_{l+k}:

- **Solve.**
  Does not apply, as the topmost element in the execution stack is not a built-in constraint.

- **Activate.**
  Does not apply, as the topmost element in the execution stack is an occurrenced CHR constraint.

- **ReActivate.**
  Does not apply, as the topmost element in the execution stack is an occurrenced CHR constraint.

- **Drop.**

  - This is the only rule that applies to state $\sigma_{l+o+1-j}$ as $o + 1$ is not a valid occurrence for c. It realizes the transition $\sigma_{l+o-j+1} \rightarrowtail \sigma'_l$.
  - It does not apply to states $\sigma_l, ..., \sigma_{l+o-j}$ since the occurrences $j..o$ do appear in program P.

- **Simplify.**
  Assume this transition rule applies. Then $S_l = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus R$ for some H_1, H_2, H_3 and R such that for some rule r in P:

  $$r \quad @ \quad H'_1 \backslash H'_2, d_j, H'_3 \Leftrightarrow g|C$$

  there exists a matching substitution $\theta \equiv (chr(H_1) = H'_1 \wedge c = d \wedge chr(H_2) = H'_2 \wedge chr(H_3) = H'_3)$ and $\mathcal{D} \models B_l \rightarrow \exists_{B_l}(\theta \wedge g)$.

Now consider the initial subsequence of transitions

$$\sigma \rightarrowtail \sigma_1 \rightarrowtail ... \rightarrowtail \sigma_d$$

of the transition sequence $\sigma \rightarrowtail \ldots \rightarrowtail \sigma_l$ where

$$\sigma_d = \langle X +\!\!+ [c\#i : j|Y], S_d, B_d, T_d \rangle_{n_d}$$

is the first state of this form. Since the execution stack behaves as a stack, this intermediate state must occur before σ_l is reached or it may coincide with σ_l if X is empty. The sequence $\sigma \rightarrowtail \ldots \rightarrowtail \sigma_{d-1}$ must contain intermediate states where the execution stack is $[c\#i : m|X] +\!\!+ [c\#i : j|Y]$ with $m = 1..o$. For the state σ_m with $m = k$:

- Either the **Simplify** rule applies. But then $c\#i$ would be removed from the store and never reoccur in it. This is contrary to the assumption that $c\#i \in S_l$.
- Or the **Simplify** rule does not apply, because:
  - At least one of the partner constraints in H_1, H_2 or H_3 is not present in the constraint store in that state. Because all partner constraints are present in the constraint store S_l, they must have been added in between σ_m and σ_l.
    Now consider the state where the last partner constraint has been added to the constraint store. From that state on all partner constraints are present in the constraint store, since they still are present in state σ_l.
    Hence, the last partner constraint must have given rise to a state before σ_l where the topmost element of the execution stack is an occurrenced version of that constraint with the same occurrence in rule r as it has in state σ_{l+k}.
    For that state the rule would either apply, but this would remove $c\#i$ and is contrary to our initial assumption. Otherwise it does not apply because the guard is not entailed or a matching substitution is missing. It is proven in the next case why this is impossible as well. Hence, this situation cannot occur.
  - All partner constraints are present in the constraint store, but the guard or matching substitution are not entailed by the built-in constraint store B_m.
    Because in subsequent states only conjuncts are added to the built-in constraint store, it must be that $B_l = B_m \wedge B_{new}$ with B_{new} a conjunction of built-in constraints.
    Since B_l entails the guard and matching, while B_m does not, some of the conjuncts in B_{new} are required for the entailment. Now,

$B_{new} = B_{new,1} \wedge B_{new,2}$ where all built-in constraints in $B_{new,1}$ have been added before those in $B_{new,2}$ and $B_m \wedge B_{new,1}$ does entail both guard and matching substitution. It is obvious that such a $B_{new,1}$ and $B_{new,2}$, with $B_{new,1} \neq true$ but potentially $B_{new,2} = true$.
Now consider the built-in constraint c_b that is a conjunct of $B_{new,1}$ and has been added the last. This addition is only possible via the **Solve** rule. The **Solve** rule adds all affected constraints in the constraint store, to the execution stack. At least one of the constraints in H_1, H_2, H_3 or $c\#i$ is so added to the execution stack. If $c\#i$ is so added, this violates the assumption that $c\#i$ appears at the top of the stack in a state between σ and σ_l. If one of the partner constraints is added, then, similar to the previous case, it will consider the **Simplify** transition rule for program rule r. All the necessary constraints are present in the constraint store and both guard and matching constraint are present. Hence, the **Simplify** will apply and remove $c\#i$ from the constraint store. However, this is contrary to the assumption that $c\#i \in S_l$. Hence, this situation cannot occur.

Hence, it is not possible for the **Simplify** rule to apply in any state σ_{l+k}.

- **Propagate.**
  Similar argument as for the **Simplify** transition.

- **Default.**
  Since the other rules do not apply to σ_{l+k} for $k = 0..(o - j)$, this rule does apply. It brings state σ_{l+k} into state σ_{l+k+1}.

Hence, the transition $\sigma_l \rightarrowtail \ldots \rightarrowtail \sigma'_l$ is the only possible one and none of the components in the intermediate states, except the execution stack, differ throughout.

□

5.4.2 Prolog's Solve Transition

It was noted only recently that the **Solve** transition (see Figure 2.2) of the refined operational semantics is actually too strict for existing CHR implementations. Indeed, consider the earlier Definition 5.1 of affected CHR constraints that are put on the execution stack in Prolog. This behavior is not sound with respect to the refined operational semantics. The following example illustrates this.

Example 5.4 Consider the following CHR program:

```
r1 @ p(X) ==> Y = a, q.
r2 @ p(X) <=> fail.
r3 @ q \ p(X) <=> true.
r4 @ q <=> true.
```

Under the refined operational semantics, the initial query `p(X)` will lead to a failed final state. First rule `r1` is applied and the goal `Y = a` causes a **Solve** transition. This transition puts the identified `p(X)` on the execution stack, which proceeds with rule `r2` and causes failure.

However, in Prolog the goal `Y = a` does not put the identified version of `p(X)` on the execution stack as `p(X)` is not affected. Hence execution proceeds with `q`, which first removes `p(X)` from the store through rule `r3` and then itself through rule `r4`. The final state is a success state with an empty CHR constraint store.

As in Prolog a final state is reached that is not reachable under the refined operational semantics, clearly Prolog is not sound with respect to this refined operational semantics.

As the refined operational semantics are supposed to describe the behavior of most current CHR implementations, a new formulation of the **Solve** rule, **Solve'** given in Figure 5.1, was made to remedy the inconsistency. Note that the new formulation is more non-deterministic than the old one: only a lower and an upper bound for the set S_1 are given.

We will now show that the set S_1 of affected CHR constraints that is put on the execution stack during Prolog's **Solve** transition is correct with respect to the **Solve'** formulation. This actually shows that the **Solve'** formulation is reasonable with respect to current CHR implementations in Prolog and the reference implementation (Holzbaur and Frühwirth 1999) in particular. Recall that the built-in constraint domain $\mathcal{D}$ of Prolog is the Herbrand constraint domain $\mathcal{H}$.

Theorem 5.2 *For the set S_1 of all affected CHR constraints in an execution state of the form $\langle [c|A], S, B, T\rangle_n$ where c is a built-in constraint, it holds in Prolog that:*

1. lower bound*: For all $M = H_1 +\!\!+ H_2 \subseteq S$ such that there exists a rule*

$$r \ @ \ H_1' \setminus H_2' \iff g' \mid C'$$

and a substitution θ such that

$$\begin{cases} chr(H_1) = \theta(H_1') \\ chr(H_2) = \theta(H_2') \\ g = \theta(g') \\ \mathcal{H} \not\models B \rightarrow \bar{\exists}_M g \\ \mathcal{H} \models B \wedge c \rightarrow \bar{\exists}_M g \end{cases}$$

then $M \cap S_1 \neq \emptyset$

Solve' $\langle [c|A], S, B, T\rangle_n \rightarrowtail \langle S_1 +\!\!+ A, S, c \wedge B, T\rangle_n$ where c is a built-in constraint, and S_1 is some subset of S satisfying the following

1. *lower bound*: For all $M = H_1 +\!\!+ H_2 \subseteq S$ such that there exists a rule
$$r \ @ \ H_1' \setminus H_2' \iff g' \mid C'$$
and a substitution θ such that
$$\begin{cases} chr(H_1) = \theta(H_1') \\ chr(H_2) = \theta(H_2') \\ g = \theta(g') \\ \mathcal{D} \models \neg(B \rightarrow \bar{\exists}_M g) \wedge (B \wedge c \rightarrow \bar{\exists}_M g) \end{cases}$$
then $M \cap S_1 \neq \emptyset$
2. *upper bound*: If $m \in S_1$ then $vars(m) \not\subseteq fixed(B)$, where $fixed(B)$ is the set of variables fixed by B.

Figure 5.1: **Solve'**, the corrected version of **Solve**

2. upper bound*: If $m \in S_1$ then $vars(m) \not\subseteq fixed(B)$, where $fixed(B)$ is the set of variables fixed by B.*

Proof: We consider only rules in Head Normal Form (see Section 2.4.1).

For the purpose of the proof we also restrict ourselves to guards of the form:

$$g \equiv \exists \bar{y_1}(c_1 \wedge \exists \bar{y_2} : (c_2 \wedge \ldots))$$

with $\exists \bar{y_i}$ either is empty and $c_i \equiv a = b$ or $\bar{y_i} = y_{i,1}, \ldots, y_{i,n}$ and $c_i \equiv a = f(y_{i,1}, \ldots, y_{i,n})$ with $a, b \in vars(M)$ variables occurring in previous existential quantifiers or in the head of the rule and f a functor and $n \geq 0$. Note that the existential quantifiers in the above are only used to indicate the introduction of new variables; in practice they are omitted.

The above restricted form facilitates the proof and all guards may be rewritten to this form.

We prove the two parts of the theorem separately.

1. *Lower Bound*
Assume that for some $M \subseteq S$, as defined in the lower bound part of the theorem, it holds that $M \cap S_1 = \emptyset$. Because of the definition of M it must be that:
$$\begin{cases} \mathcal{H} \not\models B \rightarrow \bar{\exists}_M g \\ \mathcal{H} \models B \wedge c \rightarrow \bar{\exists}_M g \end{cases}$$

Because of the syntactical restriction on guards in Prolog, it holds that

$$\bar{\exists}_M g \equiv \exists \bar{y_1}(c_1 \wedge \exists \bar{y_2} : (c_2 \wedge \ldots))$$

Let us consider this guard in a left-to-right manner with respect to B.

Consider whether $\exists \bar{y_i} c_i$ is implied by B. Start with $i = 1$.

- Either it is implied. Now consider the form of c_i and y_i. Because of the restriction on the form of constraints in the guard, either $\bar{y_i}$ is empty and $c_i \equiv a = b$ or $\bar{y_i} = y_{i,1}, \ldots, y_{i,n}$ and $c_i \equiv a = f(y_{i,1}, \ldots, y_{i,n})$ with $a, b \in vars(M)$ free variables and f a functor and $n \geq 0$. In the latter case, there must be some $z_1, \ldots, z_n$ appearing in B such that $B \rightarrow a = f(z_1, \ldots, z_n)$ because the constraint is entailed. Replace all occurrences of $y_{i,j}$ with z_j in the remainder of the guard. Continue with the remainder of the guard, i.e. increase i by one and repeat.
- Stop.

Now, clearly the first part of the guard $\exists \bar{y_i} c_i$ not entailed by B must be entailed by $B \wedge c$. For $c_i \equiv a = b$, we assume at most one of both a and b is non-variable with respect to B[3]. Say a is definitely variable and b is either variable or non-variable. Because of the restriction on the form of guards in Prolog and the above procedure for any x, either a or b, which is variable with respect to B there must be a CHR constraint $d \in M$ such that $x \in term\_vars(d)$. Hence, clearly constraint d is affected and must be part of S_1. Ergo our initial assumption that $M \cap S_1 = \emptyset$ is wrong.

If both a and b are non-variable, then either their functors or arities are different, the **Solve** transition causes inconsistency and the set S_1 is of no interest, or at least for one pair of arguments $x_i = y_i$ is not satisfied in B. The argument for $\exists \bar{y_i} c_i$ may now be repeated recursively for $x_i = y_i$ (the *term_vars* property is recursively inherited).

In the case that $c_i \equiv a = f(x_1, \ldots, x_n)$ the argument is similar.

Hence in all cases it must be that $M \cap S_1 \neq \emptyset$.

2. *Upper Bound*
From the definition of S_1 and the definition of affected it follows that:

$$\forall c \in S_1 : \exists a : \mathcal{H} \models B \rightarrow a \in term\_vars(c)$$

A variable a for which $a \in term\_vars(c)$ holds in Prolog, is not fixed. For example, two different renamings $\theta_1 = \{a/1\}$ and $\theta_2 = \{a/2\}$ map a onto the integers 1 and 2 respectively without causing any inconsistency. Hence the upper bound holds for S_1.

[3]If both a and b are non-variable, the unification may be decomposed into a number of unifications for which at most one of the arguments is non-variable

□

It is possible to extend the proof to allow other tests in the guard, such as `var/1` and `nonvar/1` and calls to predicates.

Example 5.5 Consider again Example 5.4

```
r1 @ p(X) ==> Y = a, q.
r2 @ p(X) <=> fail.
r3 @ q \ p(X) <=> true.
r4 @ q <=> true.
```

With the new definition **Solve'**, the initial query `p(X)` does not necessarily lead to a failed final state. First rule `r1` is applied and the goal `Y = a` causes a **Solve'** transition. The identified constraint `p(X)` is not affected by the unification and hence does not have to be put on the execution stack.

5.5 Conclusion

In this chapter we have introduced a basic Prolog compilation schema and explained the optimizations to it in the CHR reference implementation. To the best of our knowledge this is the most comprehensive explanation of the compilation schema and the first in terms of the refined operational semantics. The presented compilation schema will serve as a basis for further discussion of implementation aspects of CHR.

Moreover, we have established the soundness of two parts of the compilation schema with respect to the refined operational semantics. Very little work has been done regarding formal soundness proofs of CHR implementations. In Section 6.3.4 we will present another soundness proof, one of an optimization that we have developed in the context of our CHR system. To the best of our knowledge ours are the first correctness proofs of their kind.

To further increase the confidence in existing CHR implementations, more formal proofs of the optimizations are needed. In particular, the interoperability of different optimizations should be studied. Such a study might reveal problematic border-line cases that only show up when multiple optimizations apply simultaneously.

We have recently started a joint project with Christian Holzbaur concerning the formulation of a more generic host language-neutral compilation schema. The generic schema will capture the control flow of the compiled CHR program in an intermediate language that can be compiled to different host languages. In this way, CHR systems for many different host languages may use the "CHR-to-intermediate language" compiler and leverage from the optimizations that will be

developed for it. However, this project is still in a very preliminary stage and it is an important part of our future work.

Chapter 6

The K.U.Leuven CHR System

6.1 Introduction

CHR has been around for several years now, but the number of CHR implementors, the variety of available implementations and the number of Prolog systems containing such an implementation were surprisingly small at the start of our involvement with CHR.

Figure 6.1 provides an historical overview of CHR implementations, from the language's conception in 1991 to the time of writing.

In the first few years, several prototype CHR systems have been developed to experiment with and illustrate the feasibility of the language. These early systems, in Sepia, its successor ECL^{i}PS^e (IC-Parc) and its derivative Sepia* (Meier), as well as in Lisp (Steele 1984) and Oz (Smolka 1995), were rather limited. For example, no more than two constrains were allowed in the head of a rule. Most of these CHR systems are no longer in use.

The first full CHR system was developed by Christian Holzbaur in co-operation with Thom Frühwirth (Holzbaur and Frühwirth 1999; Holzbaur and Frühwirth 2000). This system was first included in the SICStus 3 release (Intelligent Systems Laboratory 2003) and later in the Yap Prolog system (Santos Costa, Damas, Reis, and Azevedo 2004). It is considered as the reference implementation of CHR. It was the first system to allow an unbounded number of constraints in the heads of rules.

Interest in CHR implementations increased in the last five years. With the help of Christian Holzbaur an evolved optimizing version of his reference implementation was ported to the HAL language. Later this implementation was rewritten by Gregory Duck.

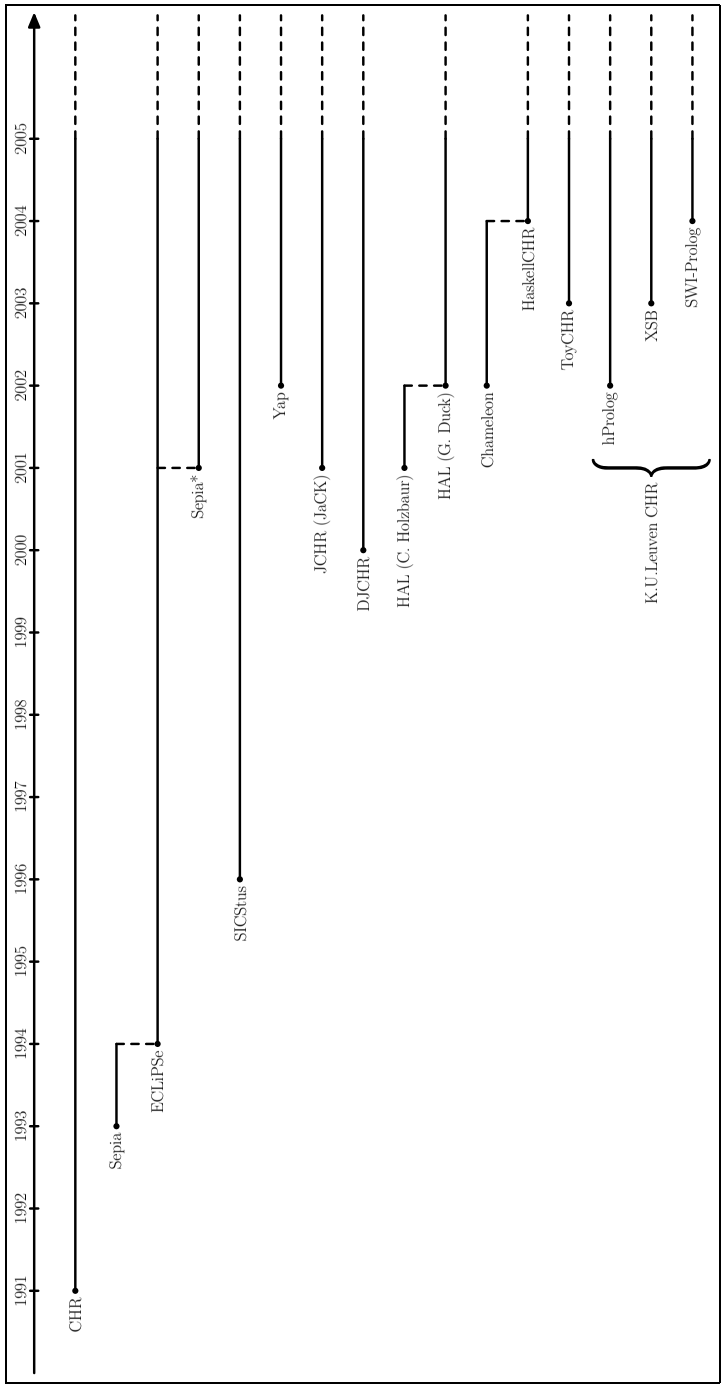


Figure 6.1: A timeline of CHR implementations

A dedicated CHR system was written by Martin Sulzmann et al. for the Haskell variant Chameleon to support the work on a customizable type system(Stuckey and Sulzmann 2005). This first Haskell implementation was later replaced by the HaskellCHR implementation by Gregory Duck. Gregory Duck also built a small new Prolog interpreter for CHR, ToyCHR, that runs in SICStus and SWI-Prolog.

JCHR is a Java implementation of CHR by Slim Abdennadher et al. that is part of the Java Constraint Kit (Abdennadher, Krämer, Saft, and Schmauss 2001). JCHR compiles a CHR program to a high-level unoptimized intermediate form that is interpreted. The interpreter does not follow the refined operational semantics, but its own instance of the theoretical operational semantics. Another CHR system for Java is the DJCHR system by Armin Wolf (Wolf 2001; Wolf 2005) which extends CHR with "justification". Justification is useful for improved non-chronological backtracking and adaptive constraint programming.

In this chapter we present our own contribution: a new CHR system for Prolog, the *K.U.Leuven CHR system*, that we have developed and extended throughout this thesis. The objective of the K.U.Leuven CHR system is threefold:

- Firstly, to provide a decent alternative to the reference implementation for Prolog. Despite the fair number of CHR implementations listed above, most have been abandoned altogether and none matches the reference implementation in efficiency.

- Secondly, to bring the current state-of-the-art in optimized compilation of CHR to Prolog. The reference implementation has changed very little over the years. While several optimizations have been developed in the context of HAL (Holzbaur, García de la Banda, Stuckey, and Duck 2005), no effort had been done to port these optimizations to Prolog.

- Thirdly and most importantly for this thesis, to serve as a basis for new research into optimized compilation, analysis and extensions of CHR.

This chapter mainly addresses the first two objectives. The implementation of our system was inspired by the reference CHR implementation of Christian Holzbaur (Holzbaur and Frühwirth 1999). Inspiration also came from several optimizations published by others (Holzbaur, García de la Banda, Stuckey, and Duck 2005). Section 6.2 outlines the general implementation of the K.U.Leuven CHR system.

The optimizations implemented in the K.U.Leuven CHR system, both those taken from related work and those we have contributed ourselves, are contained in Section 6.3.

The host system of the K.U.Leuven CHR system was originally hProlog (Demoen), but it is now also available in two major open-source Prolog systems: SWI-Prolog (Wielemaker 2004) and XSB (Warren et al. 2005). The presence in

three different Prolog systems helps realize our first objective and CHR programmers are no longer forced to choose among a limited number of Prolog systems: all major Prolog systems are covered. The two ports, to SWI-Prolog and XSB, are discussed in Section 6.4.

We verify the realization of the first two objectives with an experimental evaluation of our system. Section 6.5 compares our system to the reference implementation. Finally, Section 6.6 concludes.

The main realization of the third objective, and our most important scientific contributions, are covered in the next three chapters. As such, this chapter serves as a proper background.

6.2 Implementation

In this section we describe the main implementational aspects of the K.U.Leuven CHR system. Initially the system was written for the hProlog system (Demoen). hProlog is based on dProlog (Demoen and Nguyen 2000) and intended as an alternative backend to HAL (Demoen, García de la Banda, Harvey, Marriott, and Stuckey 1999) next to the current Mercury (Somogyi, Henderson, and Conway 1996) backend. The initial intent of the implementation of a CHR system in hProlog was to validate the underlying implementation of dynamic attributes (Demoen 2002).

The K.U.Leuven CHR system consists of two parts:

- The *runtime* library is strongly based on the SICStus CHR runtime written by Christian Holzbaur.
- The *preprocessor* compiles embedded CHR rules in Prolog program files into Prolog code. The compiled form of CHR rules is similar to that described in Section 5.2.

The advantage of the runtime code is that it is generic and can be reused for all CHR programs. However, this genericity also has its price: runtime overhead. In the evolution of the system, more and more tasks have moved away from the runtime library to specialized code generated by the compiler. This specialization process is discussed in more detail in Section 6.3.

6.3 Optimizations

This section lists the noteworthy optimizations implemented in the K.U.Leuven CHR system. Section 6.3.1 contains the optimizations that were published in related work. Our own contributions, code specialization for ground constraints, hash table constraint stores and anti-monotonic delay avoidance, are discussed in Section 6.3.2, Section 6.3.3 and 6.3.4 respectively.

6.3.1 Related Work on Optimizations

Join Ordering In the general compilation schema in Section 5.2 the ordering of partner constraint lookups for a particular active occurrence is the left-to-right ordering of these partner constraints in the rule head. However, the refined operational semantics does not specify any particular ordering, so we are free to chose any ordering.

In fact, it may have an important impact on the overall efficiency of the program what ordering is chosen. The lookup of partner constraints itself is a typical enumeration problem from constraint programming. For every partner constraint there may be many candidates and all the partner constraints together with the active constraints have to satisfy the matching and the guard. The nested lookup of partner constraints determines the search tree of this problem. Every partner constraint corresponds with a level in the search tree: the active constraint is the root of the tree, the first partner constraint corresponds with the first level, ..., the last partner constraint corresponds with the deepest level. Every edge from level $i-1$ to level i selects a particular candidate for partner constraint i. In every leaf a candidate has been selected for every partner constraint, so the guard and head matching can be verified to see whether the rule is applicable to that combination of constraints. The ordering of partner constraint lookups clearly determines the shape of the search tree.

As already explained in Section 5.2.3 lookup of partner constraints is either done by a linear search in a global list of all constraints or, given a variable that occurs in the constraint, in a list of constraints containing that variable (this is realized with attributed variables). The latter is potentially much more efficient as the number of constraints with a particular variable in it may be much smaller than the number of all constraints.

Now, the head matching and equality constraints in the guard of the rule imply identical structures in some (parts of) arguments of constraints. If at runtime (part of) one such argument contains a variable, that variable should also appear in other constraints with the same structure. Consider a level in the search tree where the first partner constraint that shares a particular structure has been selected. To lookup other partner constraints with the same shared structure, a variable-based lookup can be done if the candidate for the first partner constraints has a variable in the shared structure. Clearly, this variable-based lookup consists of a pruning of the search tree when compared with the global list-based lookup.

This suggests a heuristic for ordering the partner constraint lookups. One should try to maximize those of constraints in an ordering that share a structure with one of their predecessors in the ordering or with the active constraint. This heuristic has first been formulated in (Holzbaur, García de la Banda, Stuckey, and Duck 2005) and we have implemented it in K.U.Leuven CHR system.

Late Storage In Section 5.3 we have mentioned already an optimization to the general compilation schema to postpone constraint storage. In (Holzbaur, García de la Banda, Stuckey, and Duck 2005) a stronger pre-analysis is sketched for late storage optimization that also postpones storage past some propagation rules. We propose an even stronger late storage analysis based on abstract interpretation. It is covered in Section 7.4.

Never Stored A rule of the form

$$c \texttt{ <=> } \ldots$$

always removes constraints of the form c from the constraint store. When all the arguments of c are different variables, i.e. $c = p(X_1, \ldots, X_n)$, and no constraint p/n needs to be stored before this rule (thanks to late storage), then p/n is considered *never stored*. If a never stored constraint appears in the head of a rule, no code needs to be generated for the occurrences of the other constraints in the head of the rule. Such an occurrence of another constraint would not be able to find a never stored constraint in the constraint store. Never stored constraints together with the above optimization are described in (Holzbaur, García de la Banda, Stuckey, and Duck 2005).

In K.U.Leuven CHR system we also detect never stored constraints and apply the above optimization. In addition, for constraints for which never even a constraint suspension is created the `ID` argument in all predicates is omitted altogether as it serves no function.

Continuation Optimization In (Holzbaur, García de la Banda, Stuckey, and Duck 2005) Duck et al. sketch an optimization to the simple sequential succession of occurrences. They propose to both optimize failure and success continuations of occurrences. If a particular occurrence fails to apply, it may be possible to prove that the next occurrences will fail too. Hence, the next occurrences may be skipped over. A similar observation is possible if an occurrence succeeds to apply. Duck et al. have also implemented a weak prover to enable these optimizations.

A much stronger implementation in the K.U.Leuven CHR system and a more formal treatment with correctness proof are given in (Sneyers, Schrijvers, and Demoen 2005b). This work also contains a prover and optimization to remove redundant guards.

6.3.2 Ground Constraints

The CHR language contains one aspect that is intended exclusively for logical variables that may be constrained: the re-activation of CHR constraints. This aspect is captured by the `Solve` and `ReActivate` rules in the refined operational semantics.

However, not all kinds of CHR programs deal with logical variables. Some deal with ground constraints only and the variable support causes needless overhead for them. For example, the `gcd` program of Example 2.1 deals with ground constraints only.

The implementation of (Holzbaur, García de la Banda, Stuckey, and Duck 2005) does not have any support for logical variables and thereby no overhead for ground constraints. Moreover, it does not specify in what way the general compilation schema may be optimized for ground constraints.

In K.U.Leuven CHR system however, we do want to support the full range of CHR applications, both with and without variables. In order to avoid the needless overhead for ground constraints, the compiler specializes the general compilation schema for them.

The K.U.Leuven CHR system requires static groundness information of the form $p(g_i, \ldots, g_n)$ where g_i may be either $+$ or $?$. $g_i = +$ means that the ith argument of all constraints p/n is at all times ground and $g_i = ?$ means that nothing may be assumed. This static groundness information may either be derived through analysis or through manual specification. In Section 7.5 we present a preliminary groundness analysis that would serve the purpose. Manual specification is already fully supported.

The following minor and major optimizations are applied for ground constraints. Most of these optimizations are based on the observation that a ground constraint is never triggered.

- In combination with late storage, the suspension variable in the general schema is only instantiated from the point where it is created.

  This observation allows the specialization of conditional kill operations before the point of allocation (see Late Storage on p.81):

  ```
  ( var(ID) ->
          true
  ;
          kill(ID)
  )
  ```

  to simply `true`, i.e. omitted altogether.

- Equality tests in guards may be turned into full unifications as these behave the same for ground constraints. However, unifications are often implemented more efficiently. In addition, the K.U.Leuven CHR system moves unifications at the start of a clause's body into the head of that clause when possible. Many Prolog systems use indexing techniques to speed up such unifications in the heads of clauses.

- No continuation goal is constructed for ground constraints (see Section 5.2.2).

- No variables have to be looked for in a ground constraint to associate the constraint suspension with. See also Section 6.3.4 for a similar optimization to particular non-ground constraints.
- No propagation history needs to be maintained for particular propagation rules whose head constraints are all ground. Ground constraints consider an occurrence in a propagation rule only once. Moreover, if none of the head constraints of the propagation rule *observes* any of the other head constraints at an earlier occurrence, then the propagation history may be omitted. The latter restriction ensures that at most one of the head constraints may actively try the propagation rule while all the other head constraints are present in the constraint store. See Section 7.4.1 and further for a definition of the observation property and an analysis to derive it.

In addition, in Section 6.3.3 we present hash table constraint stores, that are only usable for ground constraints.

An Example of Compiled Code

Consider the following CHR program that computes the sum of a list of integers:

```
sum([I|Is],Sum) <=> sum(Is,PartialSum), Sum is I + PartialSum.
sum(_,Sum) <=> Sum = 0.
```

Under the general compilation schema of the previous chapter on the one hand, the generated code for this simple CHR program would be:

```
sum(List,Sum) :-
        sum_occurrence_1_2(List,Sum,ID).

sum_occurrence_1_2(List,Sum,ID) :-
        nonvar(List),
        List = [I|Is],
        !,
        ( var(ID) ->
                true
        ;
                kill(ID)
        ),
        sum(Is,PartialSum),
        Sum is I + PartialSum.
sum_occurrence_1_2(_,Sum,ID) :-
        !,
        ( var(ID) ->
                true
```

```
        ;
                kill(ID)
        ),
        Sum = 0.
sum_occurrence_1_2(List,Sum,ID) :-
        sum_drop(List,Sum,ID).

sum_drop(List,Sum,ID) :-
        make_id_sum(List,Sum,ID),
        actually_insert_sum(List,Sum,ID).
```

With the groundness declaration `sum(+,?)` the K.U.Leuven CHR system on the other hand generates the following Prolog code:

```
sum([I|Is],Sum) :- !,
         sum(Is,PartialSum),
         Sum is I + PartialSum.
sum(_,0).
```

This extremely compact code is generated thanks to the various optimizations driven by the groundness information and the *never stored* property of the `sum/2` constraints.

In (Sneyers, Schrijvers, and Demoen 2005b; Sneyers, Schrijvers, and Demoen 2005a) the K.U.Leuven CHR system is extended with type declarations and an analysis to get rid of head matchings using these type declarations. This extension generates the same Prolog code as above, even if the second rule of the CHR program is written as:

```
sum([I|Is],Sum) <=> sum(Is,PartialSum), Sum is I + PartialSum.
sum([],Sum) <=> Sum = 0.
```

where the type of the first argument of `sum/2` is a list of integers.

Observe how close the generated Prolog is to the CHR code and to idiomatic Prolog code with the same behavior:

```
sum([I|Is],Sum) :-
         sum(Is,PartialSum),
         Sum is I + PartialSum.
sum([],0).
```

It is reported in (Sneyers, Schrijvers, and Demoen 2005a) that the Prolog code generated for this program using the groundness declaration is about 2.7 times as fast as the code without such declaration.

6.3.3 Hash Table Constraint Stores

The CHR constraint store implementation of the general compilation schema, explained in Section 5.2.3, provides very fast lookup of constraints in which a known variable appears: the constraints are directly stored in an attribute on the variable.

However, attributes cannot be used with non-variables, so they provide no means to efficiently look up of ground constraints. The generic schema instead uses a global unordered list of all constraints in which it is possible to lookup in worst-case time linear in the number of constraints.

In (Holzbaur, García de la Banda, Stuckey, and Duck 2005) a more efficient data structure is proposed for lookup of ground constraints, a 234-tree, that allows for logarithmic worst-case time lookup.

Here we propose the use of an even better data structure: a hash table. A hash table allows for amortized time constant in the number of elements not only for lookup, but also for insertion and deletion. Our implementation of hash tables in Prolog uses a term as an array: the ith argument of the term is the ith entry in the array. The non-standard built-in `setarg/3` is used to update the array in a backtrackable manner.

Our hash table is *dynamic* in nature. It is initialized to a small size and whenever the load exceeds the threshold, it is doubled in size. Doubling in size means replacing the term that is stored in a global variable with a new term with double the arity and rehashing all entries of the old term to the new term.

The *hash function* $h(T)$ used to map terms T, constraints in our case, to entry numbers in the array is:

$$h(T) = (h_t(T) \bmod s) + 1$$

where s is the size of the array and $h_t(T)$ is a function that maps terms to integers. The function $h_t(T)$ should be chosen such that it is hard to find two terms that map onto the same integer value. We have used the $h_t(T)$ function of SWI-Prolog, implemented by the `term_hash/2` predicate.

To resolve *hash collision*, i.e. two terms hashing to the same array entry, we use buckets. This means that an array entry is not a single term, but a list of terms.

Experimental Evaluation: Union-Find

To experimentally validate the derived complexity derived in Section 4.7.2, we have run the CHR program in SWI-Prolog (Wielemaker 2004) using our system.

By adding the appropriate mode declarations to our program, the system establishes the groundness of shared variables and uses hash tables as constraint stores.

By initializing the hash tables to the appropriate sizes and choosing the used constants appropriately, it is possible to avoid hash table collisions. Then, the hash tables essentially behave as arrays just as in the typical imperative code and the assumptions about the constraint store made in Section 4.7.2 are effectively realized.

In contrast, the first and de facto standard CHR system, available in SICStus (Intelligent Systems Laboratory 2003), does not provide the necessary constant time operations. While it does have constant lookup time for all constraint instances of a particular constraint that contain a particular variable, it does not distinguish between argument positions. Hence, the lookup of `root(X,R)` can be done in constant time given `X`, but the lookup of `X ~> Y` is proportional to the number of `~>` constraints `X` appears in. If `X` is a node with K children, then it will be $\mathcal{O}(K)$. Moreover, while the insertion of a constraint instance is $\mathcal{O}(1)$, deletion is $\mathcal{O}(I)$, where I is the total number of instances of the constraint.

The queries we use in our experimental evaluation consist of N calls to `make/1`, to create N different elements, followed by N calls to `union/2` and N calls to `find/2`. The input arguments of the latter two are chosen at random among the elements. Even the SICStus CHR system exhibits near-linear behavior for a random set of `union` operations. So we also consider a contrived set of `union` operations: disjoint trees of elements are unioned pairwise until all elements are part of the same tree. Figures 6.2 and 6.3 show the runtime results for SICStus and SWI-Prolog. It is clear from the figure that SICStus does not show the optimal quasi-linear behavior anymore which is still observed in SWI-Prolog.

We also compare the above two cases to the case where the hash tables are not initialized to a large enough size, but instead double in size and rehash each time their load equals their size. While individual hash table operations no longer take constant time, on average they do (Cormen, Leiserson, and Rivest 1990), which is sufficient for our complexity analysis. This is confirmed by experimental evaluation (see Figure 6.4).

The above comparisons illustrate that it is vital for efficiency to use a CHR system with the proper constraint store data structures. To the best of our knowledge, the K.U.Leuven CHR system is currently the only system that provides hash table-based indexing constraint stores.

6.3.4 Anti-monotonic Delay Avoidance

In this section we summarize the *anti-monotony-based delay avoidance* optimization technique for CHR programs that we published in a technical report (Schrijvers and Demoen 2004a). It is based on static analysis and aims at avoiding the rechecking of the program rules for constraints when it is unnecessary.

The static analysis determines what argument positions of constraint symbols are anti-monotonic in all the guards in the program. Assume that a guard does not succeed. Then an argument of a constraint involved in the rule of the guard is

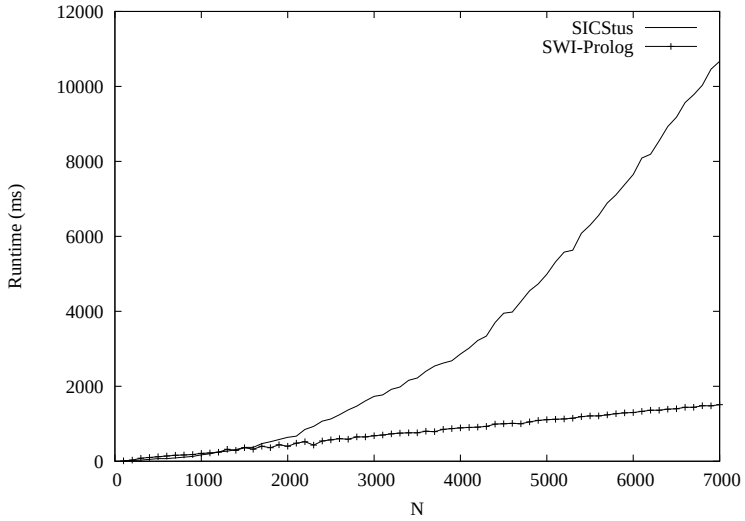


Figure 6.2: Observation of behavior for contrived unions: SICStus and SWI-Prolog array constraint stores

anti-monotonic in that guard, if further constraining that argument does not make the guard succeed. Typical examples for Prolog are guards that do not mention the argument or `var/2` tests.

As already mentioned in Section 5.2.3, in the general compilation schema, the constraint put on the execution stack are selected as follows. For every constraint *all* variables in it get the constraint's suspension associated as an attributed variable. When any of the variables is unified, the **Solve** transition selects that variable's associated constraints to be put on the execution stack.

Based on the analysis of anti-monotonic arguments, the generic association operation between variables and constraints is replaced by specialized operations, one for each constraint. Such a specialized operation for a constraint only considers arguments that are not anti-monotonic. This avoids the triggering of the constraint when any of the variables in anti-monotonic arguments is unified.

A more extensive and formal treatment, together with a correctness proof, is given in (Schrijvers and Demoen 2004a).

6.4 Ports

An initial version of K.U.Leuven CHR system was written completely in hProlog using standard Prolog code with a small number of non-standard built-ins. More

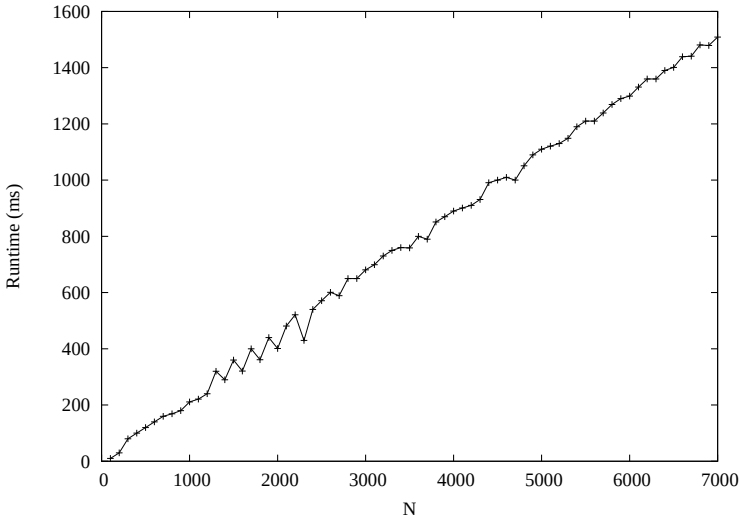


Figure 6.3: Observation of behavior for contrived unions: Detail of Figure 6.2: SWI-Prolog array constraint store

evolved versions of the system were partially written in CHR and currently the core of the compiler, not counting several auxiliary libraries, consists of almost 6,000 lines of code including 160 CHR rules.

Due to the limited number of non-standard built-ins used, porting the K.U.Leuven CHR system to other Prolog systems is relatively easy. In the course of this thesis, two such ports have actually been done and they are described in below.

6.4.1 XSB

XSB (Warren et al. 2005) is a Prolog system best known for its tabled execution extension, that allows for more succinct programs in various application domains. See Chapter 8 for our work on integrating the K.U.Leuven CHR system with XSB's tabled execution mechanism, which was the motivation for our port.

Little difficulty was experienced while porting the preprocessor and runtime system from hProlog to XSB. The main problem turned out to be XSB's overly primitive pre-existing interface for attributed variables: it did not support attributes in different modules. Moreover, the actual binding of attributed variables was not performed during the unification, but it was left up to the programmer of the interrupt handler (see Section 5.2.3). This causes unintuitive and unwanted be-

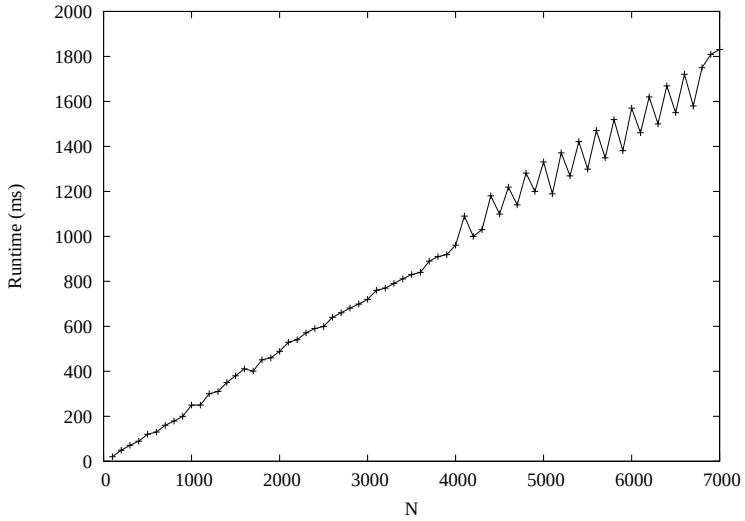


Figure 6.4: Observation of behavior for contrived unions: SWI-Prolog Hash table constraint store

havior in several cases: while the binding is delayed from unification to interrupt handling, other code can be executed in between that relies on variables being bound, e.g. arithmetic. Due to these problems with the available XSB attributed variables, it was decided to model the attributed variables interface and behavior more closely to that of hProlog. This facilitated the porting of the CHR system considerably.

The global variables interface, needed for the CHR constraint store (see Section 5.2.3), was implemented on top of a newly added single global variable that resides at the bottom of XSB's heap.

6.4.2 SWI-Prolog

SWI-Prolog (Wielemaker 2004) is a rather popular Prolog system with a large user base, a rich set of libraries and tools and a focus towards practical applications. However, support for constraint logic programming is an important aspect missing from SWI-Prolog's portfolio of features. Our experience with porting the K.U.Leuven CHR system system to XSB and CHR's focus on constraint solvers made a port to SWI-Prolog an attractive solution to remedy the lack of CLP support.

Jan Wielemaker, SWI-Prolog's lead developer, has adopted the attributed and global variable interfaces described in Section 5.2.3 with the help of Bart Demoen.

With these built-ins in place, no noteworthy obstacles were encountered during the port.

In the spirit of SWI-Prolog's user friendly environment, the CHR compiler was tightly integrated with the `term_expansion/2` based preprocessor, so as to exploit SWI-Prolog's source-code management and to retain source information. We also added a CHR debugger which hides the underlying generated Prolog code from the user.

6.5 Experimental Evaluation

6.5.1 Benchmarks

In this Section we evaluate the performance of our CHR system on ten benchmarks which are available from (Schrijvers 2005). These ten benchmarks are:

bool Addition of two 60,000 bit numbers with full-adder implemented in terms of boolean constraints.

fib Naive recursive computation of the 22 first fibonacci numbers.

fibonacci Efficient recursive computation of the 1,000 first fibonacci numbers using memoing. The fibonacci numbers are represented as floats.

leq Constraint solving using less-than-or-equal-to constraints on a ring of 60 variables. The constraints are $\bigwedge X_i \leq X_j$ for $1 \leq i \leq 60$ and $j = (i \mod 60) + 1$.

mergesort Sorting of 32 integers, repeated 10 times.

primes Computation of all prime numbers smaller than 2,500.

uf Application of the naive union-find program on the benchmark described in Section 6.3.3 with 1,000 elements.

uf_opt Application of the optimal union-find program on the benchmark described in Section 6.3.3 with 1,000 elements.

wfs Computation of the well-founded semantics of a small logic program, repeated 200 times.

zebra Computation of the solution to the well-known zebra puzzle using a naive finite domain constraint solver, repeated 10 times.

6.5.2 Systems Comparison

We compare the performance of the K.U.Leuven CHR system with that of Christian Holzbaur's reference implementation on their respective Prolog systems.

The following factors influence performance:

- Firstly, we expect the outcome to be mostly determined by the relative performance difference on Prolog code as the CHR rules are compiled to Prolog. For plain Prolog benchmarks, we have found average runtimes of 76.7 % for Yap, 80.0 % for hProlog, 143.1 % for XSB and 358.5 % for SWI-Prolog. These times are relative to SICStus. See Appendix B for some experimental measurements of relative performance.
- Secondly, the results may be influenced by the more powerful optimizations of our CHR preprocessor.
- Thirdly, the low-level implementation and representation of attributed variables differs between the systems. The standard constraint store of CHR is represented as an attributed variable and it may undergo updates each time a new constraint is imposed or a constraint variable gets bound. Hence, the complexity and efficiency of accessing and updating attributed variables may easily dominate the overall performance of a CHR program if care is not taken. Especially the length of reference chains has to be kept short and nearly constant, as otherwise accessing the cost of dereferencing the global store may easily grow out of bounds.

Table 6.1 shows the results for the benchmarks. All measurements have been made on an Intel Pentium 4 2.00 GHz with 512 MB of RAM. Timings are relative to SICStus and do not include garbage collection time. The Prolog systems used are SICStus 3.12.0 and Yap 4.4.4 with the CHR reference implementation on the one hand and hProlog 2.4.11-32, SWI-Prolog 5.5.8 and XSB 2.6.1 with the K.U.Leuven CHR system on the other hand. Because we wish to measure performance effects independent of memory management issues, we do not include garbage collection times.

We see that the relative performance difference between SICStus and Yap is more or less the same for both CHR and plain Prolog. On the other hand, the performance difference between hProlog and SICStus is about 1.56 times larger for CHR than for plain Prolog code, both times in favor of hProlog. Similarly, the performance difference between XSB and SICStus is 1.30 times smaller, in favor of XSB. Even for SWI-Prolog there is a small improvement: the factor is about 1.06.

The timing improvements are due to various minor code generation improvements and due to the care taken in implementing the runtime predicates. The good performance of the fibonacci benchmark in the K.U.Leuven CHR system is mainly thanks to the anti-monotonic delay avoidance (see also Section 6.5.4). The early

	Christian Holzbaur		K.U.Leuven		
Benchmark	SICStus	Yap	hProlog	XSB	SWI-Prolog
bool	100.0%	106.0%	51.7%	114.0%	150.3%
fib	100.0%	63.3%	59.5%	160.7%	301.2%
fibonacci	100.0%	64.3%	22.0%	64.6%	166.7%
leq	100.0%	114.7%	75.0%	151.6%	373.2%
mergesort	100.0%	63.2%	47.8%	83.3%	419.7%
primes	100.0%	123.4%	61.4%	150.2%	463.2%
uf	100.0%	69.2%	65.0%	108.6%	499.2%
uf_opt	100.0%	73.7%	69.5%	99.8%	499.2%
wfs	100.0%	78.8%	52.9%	118.1%	367.6%
zebra	100.0%	55.8%	21.0%	50.5%	133.8%
average	100.0%	81.2%	52.6%	110.1%	337.4%

Table 6.1: Runtime performance of 8 CHR benchmarks in 5 different Prolog systems.

scheduling of cheap guards in the zebra benchmark accounts for the benchmark's good behavior in the K.U.Leuven CHR system. All in all there is no one optimization that improves all benchmarks. Rather a range of different optimizations is needed, of which only a subset is applicable to any one benchmark.

6.5.3 Ground Optimizations

Now we evaluate the effect of the optimizations for ground constraints included in the hProlog and SWI-Prolog versions of the K.U.Leuven CHR system. For that purpose we have taken of the above benchmarks those that manipulate ground constraints and have added groundness declarations to them.

Table 6.2 lists the performance of the benchmarks, relative to the performance without declarations, in both hProlog and SWI-Prolog. The results for the two union-find benchmarks show that drastic improvements can be realized. The result mainly relies on the use of hash tables as constraint stores. A similar result is obtained for the `mergesort` benchmark: the use of hash tables causes the time complexity to change from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$

For the other benchmarks, the speed-ups are less dramatic, ranging between 20% and 40% in hProlog and 5% and 65% in SWI-Prolog. The improvements are due to a combination of code specialization and hash tables.

6.5.4 Anti-monotonic Delay Avoidance

To experimentally evaluate the anti-monotony-based optimization described in Section 6.3.4, we consider the effect on the runtime of our standard set of CHR

Benchmark	hProlog	SWI-Prolog
fib	57.4%	35.7%
fibonacci	57.8%	34.4%
mergesort	11.0%	4.9%
primes	81.0%	95.7%
wfs	73.3%	76.0%
uf	2.6%	1.7%
uf_opt	3.7%	2.1%

Table 6.2: Runtime performance of 7 CHR benchmarks optimized with groundness annotations relative to unoptimized programs, in both hProlog and SWI-Prolog.

benchmarks (Schrijvers 2005) together with two variants of the `fibonacci` benchmark which differ in their first rule.

The `fibonacci` program in the standard benchmark is the most optimized one. Its first rule is:

```
r1 @ fibonacci(N,M1) # ID \ fibonacci(N,M2) <=> var(M2) |
        M1 = M2 pragma passive(ID).
```

were the `passive(ID)` pragma indicates that no code should be generated for the `fibonacci(N,M1)` occurrence.

The first rule of `fibonacci1` is, similar to example 2.5, but denormalized:

```
r1 @ fibonacci(N,M1) \ fibonacci(N,M2) <=> M1 = M2.
```

Finally, `fibonacci2` has the following first rule:

```
r1 @ fibonacci(N,M1) \ fibonacci(N,M2) <=> var(M2) |
        M1 = M2.
```

The standard `fib` benchmark differs from `fibonacci1` in that it uses a simplification rule. Because this is much more inefficient, this benchmark computes a smaller Fibonacci number.

In addition to the above standard benchmarks, we have also looked at the effect on the following CHR idiom:

```
entry(Key,Value) \ lookup(Key,Query) <=> Query = Value.
```

In the above rule, the both the `entry/2` and `lookup/2` constraints are anti-monotonic with respect to their second argument. The benchmark based on this idiom is called `lookup`: it consists of asserting and entry, immediately followed by a lookup.

Table 6.3 lists the runtime results in milliseconds of running the benchmarks in hProlog. The results clearly indicate that there is hardly any effect on the

Benchmark	Optimized/Unoptimized
bool	98.7%
leq	101.2%
mergesort	100.9%
primes	100.0%
uf	102.7%
uf_opt	102.5%
wfs	100.0%
zebra	99.3%
fib	98.4%
fibonacci	53.8%
fibonacci1	46.7%
fibonacci2	46.9%
lookup	78.0%

Table 6.3: Runtime of optimized benchmarks relative to unoptimized ones, in hProlog

majority of the benchmarks. The reason is that either no static optimization is possible or dynamically no variables occur in the constraints.

In the `fib` benchmark the optimization does have some effect, but it is not manifest. The reason is that the inherent inefficiency of the simplification rule is predominant. However, in all three variants of the `fibonacci` benchmark, the runtime is about halved by the delay-avoidance optimization. Similarly, for the `lookup` benchmark there is a noticeable speedup, about 20%.

6.6 Conclusion

In this chapter we have presented the K.U.Leuven CHR system. It is a competitive CHR system in Prolog that implements state-of-the-art CHR program optimizations (Holzbaur and Frühwirth 1999; Holzbaur, García de la Banda, Stuckey, and Duck 2005) as well as several novel optimizations: hash table constraint stores, anti-monotonic delay avoidance and specialization for ground constraints.

The K.U.Leuven CHR system increases availability of CHR systems considerably: it is available in the latest releases of hProlog (Demoen), SWI-Prolog (Wielemaker 2004) and XSB (Warren et al. 2005). It uses only a small set of non-standard built-in predicates and hence it is fairly easy to port the system to other host languages as well.

The first overview of the K.U.Leuven CHR system was published at the First Workshop on Constraint Handling Rules (Schrijvers and Demoen 2004b). The work on anti-monotonic delay avoidance appeared in a technical report (Schrijvers

and Demoen 2004a). The evaluation of hash tables in the context of the union-find programs was included in the programming pearl accepted by the Theory and Practice of Logic Programming journal (Schrijvers and Frühwirth 2005). The port to SWI-Prolog has been presented as a poster at the Workshop on (Constraint) Logic Programming (Schrijvers, Wielemaker, and Demoen 2005) and the description of the port to XSB is part of the publications at the International Conference of Logic Programming (Schrijvers and Warren 2004) and the Colloquium on Implementation of Constraint and Logic Programming Systems (Schrijvers, Warren, and Demoen 2003).

6.6.1 Future Work

More ports of the K.U.Leuven CHR system are expected, in particular to Ciao Prolog and to SiLCC (a linear concurrent constraint programming language under development).

Some steps have already been taken in consolidating the improvements in the K.U.Leuven CHR system's compiler with new improvements to Christian Holzbaur's CHR system (Holzbaur and Frühwirth 1999). This project aims at a fully bootstrapping CHR compiler that generates optimized intermediate code. The generated intermediate code may be compiled to any desired host language. This approach will allow for easier maintenance of multiple host languages at once: any optimization to the generated intermediate code is immediately available to all.

Future work on generating optimized code will primarily focus on improving the partner lookup and on more powerful program analyses in the abstract interpretation framework that is presented in Chapter 7. However, other aspects that seem worthwhile to investigate are:

- More aggressive specialization of generated host language code.
- New types of constraint stores. Hybrid forms of currently available constraint stores should be able to perform better in a wider range of circumstances. Another useful line of research seems the support for more specialized constraint stores that function well for a small class of programs and the specialization of constraint stores to specific programs.
- Ideas from other rule-based languages, such as the Rete algorithm (Forgy 1982) used in production rule systems (see also Section 2.5.1).
- Exploitation of ω_t confluence. Programs that are confluent with respect to the theoretical operational semantics need not necessarily be executed using the refined operational semantics. Analyses or heuristics may be used to automatically choose execution orders that are different from those in the refined semantics.

Chapter 7

Abstract Interpretation for CHR

7.1 Introduction

Although the CHR language exists for more than ten years and has a reasonable reference implementation in SICStus Prolog (Intelligent Systems Laboratory 2003), the number of people involved in optimized compilation of and program analysis for CHR has been limited until the recent appearance of new CHR systems (Holzbaur, García de la Banda, Stuckey, and Duck 2005; Schrijvers and Demoen 2004b). The need to communicate and compare between different CHR systems has resulted in the formulation of the more deterministic refined operational semantics (Duck, Stuckey, García de la Banda, and Holzbaur 2004) shared among CHR compilers.

Apart from the common semantics to be implemented by CHR compilers, there is also a need to formalize program analyses. As the complexity of CHR compilers increases we need a better understanding of current analyses and ways to extend and combine them. Most of the currently available analyses were formulated in an ad hoc way and very few formal proofs of correctness were constructed.

Abstract interpretation (Cousot and Cousot 1977) is a general methodology for program analysis by abstractly executing the program code. Abstract interpretation provides a remedy for the current difficulties in correctly analyzing CHR programs, and should enable optimizing CHR compilers to realize more complex analyses.

In this chapter we bring the general methodology of abstract interpretation to CHR: we formulate an abstract interpretation framework over the refined denotational semantics of CHR. The formulation of an abstract interpretation framework is non-obvious since the framework needs to handle the highly non-deterministic

execution of CHRs. The use of the framework is illustrated with two instantiations: the CHR-specific *late storage* analysis and the more widely known groundness analysis. In addition, we discuss optimizations based on these analyses and present experimental results.

The rest of this chapter is structured as follows. Section 7.2 presents the refined denotational semantics of CHR that will be abstractly interpreted. The general abstract interpretation framework is then defined in Section 7.3. Two instances of the framework, late storage analysis and groundness analysis, illustrate the framework in Sections 7.4 and 7.5 respectively. The implementation and experimental evaluation of these analyses are reported on in Section 7.6. We conclude in Section 7.7.

7.2 The Refined Denotational Semantics ω_d

In this section we present the refined denotational semantics ω_d. It is a variant of the refined operational semantics ω_r (see Section 2.4.3) designed to make the formulation of analyses simpler.

We introduce the refined denotational semantics for CHR to make the number of abstract goals to be considered finite. For the same reason logic programs are not directly analyzed in terms of their derivations-based operational semantics, but instead a call-based denotational semantics was introduced (see e.g. (Marriott, Søndergaard, and Jones 1994)).

It is shown in (Duck, Schrijvers, and Stuckey 2004) that an intermediate form between ω_r and ω_d, a call-based refined operational semantics ω_c, and ω_r are equivalent. It should be straightforward to establish that ω_c and ω_d are equivalent.

The main difference between the ω_d and ω_r semantics lies in their formulation. The transition system of ω_r linearizes the dynamic call-graph of CHR constraints into the execution stack of its execution states. In ω_d constraints are treated as procedure calls: each newly added active constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires other CHR constraints may be executed as subcomputations and, when they finish, the execution returns to finding rules for the current active constraint. The latter semantics is much closer to the procedure-based target languages, like Prolog and HAL.

We believe this closeness to target languages makes the ω_d semantics much more suitable for reasoning about optimizations. After all, optimizations are typically formulated at the level of the generated code in the target language.

We will use a numbered notation for CHR programs so that it is easier to refer to occurrences of constraints: to every head constraint we add its occurrence number in brackets as a subscript.

Example 7.1 The numbered version of the `gcd` program of Example 2.1 is

```
gcd(0)[1] <=> true.
gcd(I)[3] \ gcd(J)[2] <=> J >= I | K is J - I, gcd(K).
```

The rest of this section is structured as follows. In Sections 7.2.1 and 7.2.2 we present the execution state and semantic function of ω_d. Section 7.2.3 illustrates the semantics on an example.

7.2.1 Execution State of ω_d

Formally, the execution state of the refined denotational semantics is represented by the tuple $\langle G, A, S, B, T\rangle_n$. The different components of the execution state are defined in a similar way as those of the ω_r semantics: The execution stack of ω_r is more or less split into the *goal* and *execution stack* components of the ω_d semantics. The goal corresponds to the current "procedure call", whereas the execution stack corresponds to the "ancestor calls". Due to the use of recursion in the semantic function (defined in Section 7.2.2) it is not necessary to maintain all the information of ω_r's execution stack in either ω_d's goal or ω_d's execution stack.

The *goal* G is either a sequence of (possibly occurrenced and identified) CHR constraints and built-in constraints or just a single constraint. If it is a single constraint, that constraint is called the *active constraint* and it corresponds to the active constraint of the ω_r semantics. The *execution stack* A is a sequence of constraints c, identified CHR constraints $c\#i$ and occurrenced identified CHR constraints $c\#i:j$. The remaining components are the same as in ω_r: The *CHR store* S is a set of identified CHR constraints. The *built-in constraint store* B contains any built-in constraint that has been passed to the underlying solver. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints which fired a rule, and the name of the rule itself. Finally, the *next free identity* n represents the next integer which can be used to number a CHR constraint.

We denote the domain of execution states by $\mathbf{\Sigma}$ and elements of $\mathbf{\Sigma}$ as $\sigma, \sigma_0, \sigma_1, \ldots$ Given initial goal G, the initial state is $\langle G, \Box, \emptyset, \mathit{true}, \emptyset\rangle_1$.

The function pp returns the *program point* of an execution state:

$$
\begin{array}{rcll}
\mathsf{pp}(\langle G, A, S, B, T\rangle_n) & = & \mathsf{pp}(G) & \\
\mathsf{pp}(c) & = & \texttt{builtin} & \text{(c built-in)} \\
\mathsf{pp}(p(x_1, \ldots, x_n)) & = & p/n & \\
\mathsf{pp}(p(x_1, \ldots, x_n)\#i) & = & p/n & \\
\mathsf{pp}(p(x_1, \ldots, x_n)\#i:j) & = & p/n:j & \\
\mathsf{pp}([c_1, \ldots, c_n]) & = & [\mathsf{pp}(c_1), \ldots, \mathsf{pp}(c_n)] &
\end{array}
$$

Traditionally a program point corresponds to a location in the program code. Also, the current program point of an execution of the program is maintained at all times in a part of the execution state called the program counter. However, in the execution of CHR the coupling between the program code and the execution states is less explicit. In many execution states it is not necessary to know the program in order to proceed.

Hence, instead of defining locations in a CHR program $\mathcal{P}$, the program points of pp relate execution states to locations in the code of the compilation schema: The program point p/n corresponds to the code for the **Activate** transition of constraint p/n and the program point $p/n : i$ corresponds to the code for occurrence i (see 5.2.1). If also programs points concerning built-in constraints are of interest, e.g. for optimizing these, the special value `builtin` should probably be replaced with a more informative value.

7.2.2 Semantic Function of ω_d

The effect of executing a CHR program on an execution state σ is to change σ into a final execution state. This effect is captured by the semantic function $\mathcal{S}$, a partial function on execution states. Given an execution state, it returns a final execution state:

$$\mathcal{S} : \mathbf{Prog} \rightarrow (\mathbf{\Sigma} \hookrightarrow \mathbf{\Sigma})$$

The definition of the semantic function is given in Table 7.1. Apart from their recursive nature, most of the cases of the semantic function correspond directly to the transition rules of ω_r. The **Simplify** and **Propagate** cases differ somewhat from the transitions of the same name in ω_r. They combine the behavior of the transitions of the same name with that of the **Default** transition of ω_r. In addition the **Simplify** case applies at once the effect of successive **Default** transitions and a final **Drop** transition when the current constraint is removed by the simplification. The **Propagate** cases differs also from the **Propagate** transition in that it applies all possible successive **Propagate** transitions at once (through a call to the $\mathcal{S}_{\text{Prop}}$ function). The **Goal** case takes care of decomposing a goal sequence into individual sequences. This case is specific to ω_d, because ω_r does not have a goal component in its execution states.

7.2.3 Example

The refined denotational semantics is illustrated on a small example program:

```
p[1]  ==>  q.
p[2], t[1]  <=>  r.
p[3], r[1] ==> true.
p[4]  ==> s.
p[5], s[1] <=> true.
```

1. Solve

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_\emptyset B \wedge c \\ \qquad \text{then} \quad \langle \Box, A, S, B \wedge c, T\rangle_n \\ \qquad \text{else} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle S_1, A, S, B \wedge c, T\rangle_n)\end{array}$$

where c is a built-in constraint and $S_1 = \mathit{solve}[\![\mathcal{P}]\!](S, B, c)$ is a subset of S satisfying the following conditions:

1. *lower bound*: For all $M = H_1 +\!\!+ H_2 \subseteq S$ such that there exists a rule $r \in \mathcal{P}$
$$r \ @ \ H_1' \setminus H_2' \iff g \mid C$$
in P and a substitution θ such that
$$\begin{cases} \mathit{chr}(H_1) = \theta(H_1') \\ \mathit{chr}(H_2) = \theta(H_2') \\ \mathcal{D}_b \not\models B \rightarrow \exists_r(\theta \wedge g) \\ \mathcal{D}_b \models B \wedge c \rightarrow \exists_r(\theta \wedge g) \end{cases}$$
then $M \cap S_1 \neq \emptyset$
2. *upper bound*: If $m \in S_1$ then $\mathit{vars}(m) \not\subseteq \mathit{fixed}(B)$, where $\mathit{fixed}(B)$ is the set of variables fixed by B.

The actual definition of the *solve* function will depend on the underlying solver.

2a. Activate

$$\mathcal{S}[\![\mathcal{P}]\!](\langle c, A, S, B, T\rangle_n) = \mathcal{S}[\![\mathcal{P}]\!](\langle c\#n : 1, A, \{c\#n\} \uplus S, B, T\rangle_{(n+1)})$$

where c is a CHR constraint.

2b. Reactivate

$$\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i, A, S, B, T\rangle_n) = \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : 1, A, S, B, T\rangle_n)$$

where c is a CHR constraint.

---

3. Drop

$$\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \langle \Box, A, S, B, T\rangle_n$$

where $c\#i : j$ is an occurrenced CHR constraint and there is no such occurrence j in $\mathcal{P}$.

Table 7.1: The refined denotational semantics of CHR

4. Simplify
Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H'_1 \setminus H'_2, d_{[j]}, H'_3 \iff g \mid C$$

then

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \textit{simplify-condition} \\ \quad\quad \text{then} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), A, H_1 \uplus S', \theta \wedge B, T \cup \{t\}\rangle_n) \\ \quad\quad \text{else} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j+1, A, S, B, T\rangle_n)\end{array}$$

where *simplify-condition* is that there exists a matching substitution θ such that

$$\left\{\begin{array}{l} S = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S' \\ c = \theta(d) \\ chr(H_1) = \theta(H'_1) \wedge chr(H_2) = \theta(H'_2) \wedge chr(H_3) = \theta(H'_3) \\ \mathcal{D}_b \models B \rightarrow \bar{\exists}_r(\theta \wedge g) \\ t = id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{array}\right.$$

---

5. Propagate
Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H'_1, d_{[j]}, H'_2 \setminus H'_3 \iff g \mid C$$

then

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B_k \\ \quad\quad \text{then} \ \langle \Box, A, S_k, B_k, T_k\rangle_{n_k} \\ \quad\quad \text{else} \ \ \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j+1, A, S_k, B_k, T_k\rangle_{n_k})\end{array}$$

where $\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \langle \Box, A, S_k, B_k, T_k\rangle_{n_k}$.
The auxiliary function $\mathcal{S}_{\text{Prop}} : \mathit{Prog} \rightarrow (\mathbf{\Sigma} \hookrightarrow \mathbf{\Sigma})$ is defined as:

$$\begin{array}{l}\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B \\ \quad\quad \text{then} \ \langle \Box, A, S, B, T\rangle_n \\ \quad\quad \text{else} \ \ \text{if } \textit{propagate-condition} \\ \quad\quad\quad\quad \text{then} \ \ \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S', B', T'\rangle_{n'}) \\ \quad\quad\quad\quad \text{else} \ \ \langle \Box, A, S, B, T\rangle_n\end{array}$$

where *propagate-condition* is that there exists a matching substitution θ such that

$$\left\{\begin{array}{l} S = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus R \\ c = \theta(d) \\ chr(H_1) = \theta(H'_1) \wedge chr(H_2) = \theta(H'_2) \wedge chr(H_3) = \theta(H'_3) \\ \mathcal{D}_b \models B \rightarrow \bar{\exists}_{\theta_l(r)}\theta_l(g) \\ t = id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T \\ \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), [c\#i : j|A], S \setminus H_3, B, T \cup \{t\}\rangle_n) = \langle \Box, [c\#i : j|A], S', B', T'\rangle_{n'} \end{array}\right.$$

where g and $\theta(g)$, respectively G and $\theta(G)$ are variants and $vars(g) \cap vars(\theta(g)) = \emptyset$ and $vars(G) \cap vars(\theta(G)) = \emptyset$.

6. Goal

$$\begin{aligned}
&\mathcal{S}[\![\mathcal{P}]\!](\langle \square, A, S, B, T\rangle_n) = \\
&\qquad \langle \square, A, S, B, T\rangle_n \\
&\mathcal{S}[\![\mathcal{P}]\!](\langle [c|C], A, S, B, T\rangle_n) = \\
&\qquad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B' \\
&\qquad\qquad \text{then } \langle \square, A, S', B', T'\rangle_{n'} \\
&\qquad\qquad \text{else } \mathcal{S}[\![\mathcal{P}]\!](\langle C, A, S', B', T'\rangle_{n'})
\end{aligned}$$

where $[c|C]$ is a sequence of built-in and CHR constraints and

$$\mathcal{S}[\![\mathcal{P}]\!](\langle c, A, S, B, T\rangle_n) = \langle \square, A, S', B', T'\rangle_{n'}$$

All the occurrences of constraints in the above program are annotated with their respective occurrence numbers. Starting from an initial goal `p` the application of the semantic function of the refined denotational semantics goes as follows (for brevity we omit the propagation history, denoted by $\bullet$).

Every step is annotated with the corresponding case of the semantic function. For both the simplification and propagation steps we annotate the step name with $\neg$ if the rule did not find a match.

$$\begin{aligned}
&\mathcal{S}[\![\mathcal{P}]\!](\langle p, [], \emptyset, \mathit{true}, \emptyset\rangle_1) && (\textbf{Activate}) \\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1:1, [], \{p\#1\}, \mathit{true}, \bullet\rangle_2) && (\textbf{Propagate}) \\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1:2, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) && (\neg\textbf{Simplify}) \\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1:3, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) && (\neg\textbf{Propagate}) \\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1:4, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) && (\textbf{Propagate}) \\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1:5, [], \{q\#2\}, \mathit{true}, \bullet\rangle_4) && (\neg\textbf{Simplify}) \\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1:6, [], \{q\#2\}, \mathit{true}, \bullet\rangle_4) && (\textbf{Drop}) \\
&\quad = \langle \square, [], \{q\#2\}, \mathit{true}, \bullet\rangle_4
\end{aligned}$$

For the first propagation step above, the result of the auxiliary function $\mathcal{S}_{\text{Prop}}$ is used:

$$\begin{aligned}
&\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1:1, [], \{p\#1\}, \mathit{true}, \bullet\rangle_2) \\
&\quad = \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1:1, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) \\
&\quad = \langle \square, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3
\end{aligned}$$

The second step in the evaluation of $\mathcal{S}_{\text{Prop}}$ is obtained through the evaluation of:

$$\begin{aligned}
&\mathcal{S}[\![\mathcal{P}]\!](\langle q, [p\#1:1], \emptyset, \{p\#1\}, \emptyset\rangle_2) && (\textbf{Activate}) \\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle q\#2:1, [p\#1:1], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) && (\textbf{Drop}) \\
&\quad = \langle \square, [p\#1:1], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3
\end{aligned}$$

The last propagation step in the main computation is obtained through a similar evaluation of $\mathcal{S}_{\text{Prop}}$:

$$\begin{aligned}
&\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1:4,[],\{p\#1,q\#2\},\mathit{true},\bullet\rangle_3)\\
&\quad = \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1:4,[],\{q\#2\},\mathit{true},\bullet\rangle_4)\\
&\quad = \langle\Box,[],\{q\#2\},\mathit{true},\bullet\rangle_4
\end{aligned}$$

The second step in the above evaluation of $\mathcal{S}_{\text{Prop}}$ is obtained through:

$$\begin{aligned}
&\mathcal{S}[\![\mathcal{P}]\!](\langle s,[p\#1:4],\emptyset,\{p\#1,q\#2\},\emptyset\rangle_3) && (\textbf{Activate})\\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle s\#3:1,[p\#1:4],\{p\#1,q\#2,s\#3\},\mathit{true},\bullet\rangle_4) && (\textbf{Simplify})\\
&\quad = \mathcal{S}[\![\mathcal{P}]\!](\langle\Box,[p\#1:4],\{q\#2\},\mathit{true},\bullet\rangle_4) && (\textbf{Goal})\\
&\quad = \langle\Box,[],\{q\#2\},\mathit{true},\bullet\rangle_4
\end{aligned}$$

7.3 The Abstract Interpretation Framework

In this section we present our generic abstract interpretation framework for CHR.

An abstract interpretation framework consists of:

- an abstract domain of execution states, together with an abstraction function α and a concretization function γ to translate from, respectively to concrete execution states,
- an abstract operational semantics.

Our framework is generic: it does not fully specify the abstract semantics and abstract domain, but rather imposes restrictions on actual instances that must provide a full specification. In particular, our framework formulates the abstract semantics in terms of an abstract semantic function that must be provided by instances of the framework.

In Sections 7.3.1 and 7.3.2 we discuss how a particular instance of the framework, i.e. an analysis domain, should specify its abstract state and abstract semantic function. The generic, domain-independent aspects of the abstract semantics, which are provided by the framework, are presented in Section 7.3.3. It covers how the framework applies the abstract semantic function starting from an initial state and how the framework deals with non-determinism.

7.3.1 Abstract State

Every instance of the abstract interpretation framework should define an abstract domain $\mathbf{\Sigma_a}$ of abstract states. The abstract domain $\mathbf{\Sigma_a}$ has to be a lattice with partial ordering $\preceq$, least upper bound $\sqcup$ and greatest lower bound $\sqcap$ operations.

Furthermore an abstraction function $\alpha : \wp(\mathbf{\Sigma}) \to \mathbf{\Sigma_a}$ has to be defined from a set of concrete states σ, as defined in Section 7.2.1, to an abstract state s and a concretization function $\gamma : \mathbf{\Sigma_a} \to \wp(\mathbf{\Sigma})$ from an abstract state to a set of concrete states.

Typically we only specify α and assume γ to be defined as $\gamma(s) = \{\sigma \mid \alpha(\{\sigma\}) = s\}$. Moreover, in an abuse of syntax we denote $\alpha(\{\sigma\})$ as $\alpha(\sigma)$.

As is usual, we require that (α, γ) is a *Galois connection* of $(\wp(\mathbf{\Sigma}), \subseteq)$ and $(\mathbf{\Sigma_a}, \preceq)$, i.e.

$$\forall S \in \wp(\mathbf{\Sigma}) : \forall s \in \mathbf{\Sigma_a} : \alpha(S) \preceq s \Leftrightarrow S \subseteq \gamma(s)$$

We impose an additional restriction on γ:

$$\forall s \in \mathbf{\Sigma_a} : \forall \sigma_1, \sigma_2 \in \gamma(s) : \mathsf{pp}(\sigma_1) = \mathsf{pp}(\sigma_2)$$

i.e. every abstract execution state should correspond with exactly one program point. This allows us to extend the domain of the pp function to abstract states:

$$\mathsf{pp}(s) = \mathsf{pp}(\sigma) \text{ with } \sigma \in \gamma(s)$$

The restriction is imposed for two reasons:

- to be able to associate analysis information contained in abstract states with the program points, and
- to determine whether a particular abstract state s is a final state (i.e. $\mathsf{pp}(s) = \square$).

The accurate program point information may complicate the abstract semantics somewhat, but results in more accurate analyses.

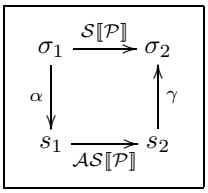
The framework will only make use of the least upper bound operation $s_1 \sqcup s_2$ on states corresponding to the same program point ($\mathsf{pp}(s_1) = \mathsf{pp}(s_2)$). Similarly, α is only applied to a set of concrete states corresponding to the same program point. Moreover, we will only explicitly define α for a single concrete state σ. The extension of α to a set S of concrete states is assumed to be:

$$\alpha(S) = \bigsqcup_{\sigma \in S} \alpha(\sigma)$$

7.3.2 Abstract Semantic Function

The abstract domain must provide an abstract semantic function $\mathcal{AS} : \mathbf{Prog} \rightarrow (\mathbf{\Sigma_a} \hookrightarrow \mathbf{\Sigma_a})$ with abstract cases AbstractSolve, AbstractActivate, AbstractReactivate, AbstractDrop, AbstractSimplify, AbstractPropagate and AbstractGoal corresponding to the cases of the concrete semantic function $\mathcal{S}$, as given in Section 7.2.2.

In order for the abstract semantic function to be a consistent abstraction of the concrete semantic function, we impose the connection depicted below:



or formally:

$$\forall S \subseteq \mathbf{\Sigma} : \{\mathcal{S}[\![\mathcal{P}]\!](\sigma) | \sigma \in S\} \subseteq \gamma \circ \mathcal{AS}[\![\mathcal{P}]\!] \circ \alpha(S)$$

7.3.3 The Generic Abstract Semantics

Here we explain the generic semantics of the framework, based on the analysis-specific implementations of the abstract domain and the abstract semantic function.

The concrete operational semantics specifies that the semantic function is applied to an initial state to obtain a final state. In the following we describe what initial state is used by the framework and how the abstract semantic function should be defined. In particular the issue of non-determinism is discussed.

Generic Initial State

For any CHR program, an infinite number of concrete initial states are possible, namely any $\langle G, [], \emptyset, \emptyset, \emptyset \rangle_1$ with G any finite list of CHR constraints and built-in constraints.

This infinite number of initial states may lead to an infinite number of abstract states, depending on the definition of α. However, in the generic framework we avoid this potential blow-up of initial states by requiring that the initial goal is a single CHR constraint c.

The requirement of a single constraint is not a restriction. It is always possible to encode a list of multiple goals $c_1, \ldots, c_n$ in this way. Namely one can introduce a fresh constraint c and a new simplification rule $c \Leftrightarrow c_1, \ldots, c_n$. This new c can then serve as the single initial goal.

Similarly, it is possible to encode arbitrary sequences of constraints, using random data generators that return values in a particular domain. For example an arbitrary sequence of a and b constraints may be encoded as follows:

```
c <=> random(X), c(X).
c(1) <=> a, c.
c(2) <=> b, c.
c(_) <=> true.
```

Here the predicate `random/1` returns in its argument a random integer. The constraint c serves as the initial goal.

The key issue is that the single goal should be representative with respect to the analysis domain for all intended uses of the CHR program. This may require intimate knowledge of both the program and the analysis domain. Hence, the developer of the analysis domain should provide guidelines regarding the choice of the initial goal. It may be possible to automatically derive a default goal from a program that captures all possible uses, although a program-specific goal would yield stronger analysis results.

Non-determinism in the Simplify Case

In the **Simplify** case of the semantic function, either (1) a matching substitution θ is found and the simplification takes place or (2) no matching substitution exists and simplification does not take place.

An abstract semantic function may not be able to decided from an abstract execution state s which of the above two alternatives applies. This is the case when $\exists \sigma_1, \sigma_2 \in \gamma(s)$ such that the first alternative applies to σ_1 and the second to σ_2.

The recommended approach in this case is to compute a least upper bound of the two alternatives in the following way:

$$\mathcal{AS}[\![\mathcal{P}]\!](s) = \mathcal{AS}[\![\mathcal{P}]\!](s_1) \sqcup \mathcal{AS}[\![\mathcal{P}]\!](s_2)$$

where

$$\alpha(\{\langle \theta(C), A, H_1 \uplus S', \theta \wedge B, T \cup \{t\}\rangle_n\} | \langle c\#i : j, A, S, B, T\rangle_n \in \gamma(s)) \preceq s_1$$

and

$$\alpha(\{\langle c\#i : j+1, A, S, B, T\rangle_n | \langle c\#i : j, A, S, B, T\rangle_n \in \gamma(s)\}) \preceq s_2$$

and θ, H_1, S' and t are defined in the **Simplify** case of $\mathcal{S}$.

Non-determinism in the Choice of Partner Constraints

While the above accounts for the non-determinism in simplification matching caused by abstraction, it does not account for the inherent non-determinism of these cases in the concrete semantics.

Namely, for a simplification case, if more than one combination of partner constraints is possible, the concrete semantics do not specify what particular combination is chosen. To account for this non-determinism the formulation of the AbstractSimplify case should capture all possible concrete possibilities. In particular, if for concrete state σ there are n different possible resulting final states $\sigma_1, \ldots, \sigma_n$, then

$$\alpha(\{\sigma_1, \ldots, \sigma_n\}) \preceq \mathcal{AS}[\![\mathcal{P}]\!](\alpha(\sigma))$$

Similarly, for a propagation transition, multiple combination transitions are possible. In addition, for a propagation transition, multiple applications are possible in a sequence. However, the order of the sequence is not specified by the concrete semantics either. Hence, an abstract propagation transition has to capture all possible partner combinations and all possible sequences in which they are dealt with.

Non-determinism in the Solve Rule

The non-determinism inherent in the concrete Solve case lies in the order of the triggered constraints as they are put on the execution stack: all possible orderings are allowed. Hence, an abstract domain has to provide an abstraction that takes into account all possible orderings.

If the abstract domain allows it, one approach would be to compute the final state s_o for each possible ordering o and to combine these final states to a single final state s as follows: $s = \bigsqcup_o s_o$.

However, this requires sufficiently concrete information about the number of triggered constraints in the abstract domain. Typically the abstract domain cannot provide any quantitative bound on the number of triggered constraints. Hence an infinite number of orderings are possible: all possible permutations of constraint sequences of any integer length.

A finite approximation of this infinite number of possibilities is to perform the following fixedpoint computation. Say $\{c_i | 1 \leq i \leq n\}$ are all the possible distinct abstract CHR constraints to trigger. Then, starting from abstract state s_0, the final state s_f after triggering all constraints in any quantity is s_k, where:

$$s_j = \bigsqcup \{ s_j^i \mid s_j^i = \mathcal{AS}[\![\mathcal{P}]\!](\mathsf{new\_goal}(s_{j-1}, c_i)) \wedge 1 \leq i \leq n \}$$

for $j > 0$ and k is the smallest integer such that $s_k = s_{k+1}$. In the above formula new_goal is the function that replaces the empty goal in a final abstract state s_{j-1} with a new goal c_i.

This generic approach is illustrated in the prototype groundness analysis, discussed in Section 7.5.

Due to its generality it may cause a huge loss of precision as well as an exponential number of intermediate states. Hence, in practice, better domain specific techniques should be studied.

For example, in the late storage analysis discussed in the next section, the worst possible abstract state is immediately obtained in the `AbstractSolve` transition, before triggered constraints are considered. Hence there is no need to actually compute the triggering of constraints. The outcome is already determined. This avoids substantial overhead.

7.4 Late Storage Analysis

In this section we illustrate the use of the abstract interpretation framework for CHR with a CHR-specific analysis: late storage. This analysis is useful in CHR compilers to enable several optimizations.

In Section 7.4.1 we define the property that the analysis derives. Next, the abstract domain and abstract semantic function of the analysis are defined in

Sections 7.4.2 and 7.4.3 respectively. Section 7.4.4 illustrates the application of the analysis on a small program.

7.4.1 The Observation Property

The aim of late storage analysis is to determine for an active CHR constraint whether it can be stored *later* rather than stored *before* its rules are searched for matching. This is done is by determining when the first possible interaction will be with the active CHR constraint.

In general it is better to store a constraint in the constraint store as late as possible. The reason is that if the constraint is deleted before it is actually stored, the overhead of insertion in and removal from the constraint store are avoided.

The refined operational semantics however dictate that a constraint is inserted in the constraint store immediately when it is at the top of the execution stack. We want to avoid this when it does not make a difference to the final state.

At the latest, a constraint that is not deleted, has to be stored after all the rules have been tried. There are however also reasons for storing a constraint early. Namely, if a rule applies, the body may observe whether the active constraint is in the constraint store or not. If the active constraint may be observed, the constraint needs to be in the constraint store. Otherwise it does not have to be in the constraint store, because its presence cannot impact the execution.

Definition 7.1 (Observed) *A constraint in the constraint store is observed, if it is triggered by a built-in constraint or if it serves as a partner constraint to an active constraint.*

To correctly define the analysis of "observation" as an abstract interpretation we have to extend the refined denotational semantics to make this visible. We will only be interested in finding the observed occurrences of constraints in the execution stack.

Denote an *observed* occurrence $c\#i : j$ by starring e.g. $c\#i : j^*$. Define

$$\begin{array}{rcll} obs(c\#i : j) & = & c\#i : j^* & \\ obs(c\#i : j^*) & = & c\#i : j^* & \\ obs([], S) & = & [] & \\ obs([c\#i : j|G], S) & = & [obs(c\#i : j)|obs(G, S)] & \text{, if } c\#i \in S \\ obs([c\#i : j|G], S) & = & [c\#i : j|obs(G, S)] & \text{, if } c\#i \notin S \end{array}$$

Only the **Solve**, **Simplify** and **Propagate** cases are affected. Basically we modify the activation stack to record which constraints have been observed by any of these transitions.

1. Solve

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B \wedge c \\ \qquad \text{then} \quad \langle \Box, A, S, B \wedge c, T\rangle_n \\ \qquad \text{else} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle S_1, obs(A, S_1), S, B \wedge c, T\rangle_n)\end{array}$$

where c is a built-in constraint and $S_1 = solve[\![\mathcal{P}]\!](S, B, c)$ is a subset of S satisfying the following conditions:

1. *lower bound*: For all $M = H_1 +\!\!+ H_2 \subseteq S$ such that there exists a rule $r \in \mathcal{P}$

$$r \ @ \ H_1' \setminus H_2' \iff g \mid C$$

in P and a substitution θ such that

$$\begin{cases} chr(H_1) = \theta(H_1') \\ chr(H_2) = \theta(H_2') \\ \mathcal{D}_b \not\models B \rightarrow \exists_r(\theta \wedge g) \\ \mathcal{D}_b \models B \wedge c \rightarrow \exists_r(\theta \wedge g) \end{cases}$$

then $M \cap S_1 \neq \emptyset$

2. *upper bound*: If $m \in S_1$ then $vars(m) \not\subseteq fixed(B)$, where $fixed(B)$ is the set of variables fixed by B.

The actual definition of the *solve* function will depend on the underlying solver.

---

4. Simplify

Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H_1' \setminus H_2', d_{[j]}, H_3' \iff g \mid C$$

then

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \textit{simplify-condition} \\ \qquad \text{then} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), obs(A, H_1 \cup H_2 \cup H_3), H_1 \uplus S', \theta \wedge B, T \cup \{t\}\rangle_n) \\ \qquad \text{else} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j+1, A, S, B, T\rangle_n)\end{array}$$

where *simplify-condition* is that there exists a matching substitution θ such that

$$\begin{cases} S = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S' \\ c = \theta(d) \\ chr(H_1) = \theta(H_1') \wedge chr(H_2) = \theta(H_2') \wedge chr(H_3) = \theta(H_3') \\ \mathcal{D}_b \models B \rightarrow \bar{\exists}_r(\theta \wedge g) \\ t = id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

---

5. Propagate

Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H_1', d_{[j]}, H_2' \setminus H_3' \iff g \mid C$$

then

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B_k \\ \quad\quad \text{then } \langle \Box, A, S_k, B_k, T_k\rangle_{n_k} \\ \quad\quad \text{else } \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j+1, A, S_k, B_k, T_k\rangle_{n_k})\end{array}$$

where $\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \langle \Box, A, S_k, B_k, T_k\rangle_{n_k}$.

The auxiliary function $\mathcal{S}_{\text{Prop}} : \mathit{Prog} \to (\mathbf{\Sigma} \hookrightarrow \mathbf{\Sigma})$ is defined as:

$$\begin{array}{l}\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B \\ \quad\quad \text{then } \langle \Box, A, S, B, T\rangle_n \\ \quad\quad \text{else if } \textit{propagate-condition} \\ \quad\quad\quad\quad\quad\quad \text{then } \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A', S', B', T'\rangle_{n'}) \\ \quad\quad\quad\quad\quad\quad \text{else } \langle \Box, A, S, B, T\rangle_n\end{array}$$

where *propagate-condition* is that there exists a matching substitution θ such that

$$\left\{\begin{array}{l} S = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus R \\ c = \theta(d) \\ \mathit{chr}(H_1) = \theta(H_1') \wedge \mathit{chr}(H_2) = \theta(H_2') \wedge \mathit{chr}(H_3) = \theta(H_3') \\ \mathcal{D}_b \models B \to \bar{\exists}_{\theta_l(r)} \theta_l(g) \\ t = \mathit{id}(H_1) +\!\!+ [i] +\!\!+ \mathit{id}(H_2) +\!\!+ \mathit{id}(H_3) +\!\!+ [r] \notin T \\ \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), [c\#i : j | \mathit{obs}(A, H_1 \cup H_2 \cup H_3)], S \setminus H_3, B, T \cup \{t\}\rangle_n) = \\ \qquad \langle \Box, [E|A'], S', B', T'\rangle_{n'} \\ E \in \{c\#i : j, c\#i : j^*\} \end{array}\right.$$

where g and $\theta(g)$, respectively G and $\theta(G)$ are variants and $\mathit{vars}(g) \cap \mathit{vars}(\theta(g)) = \emptyset$ and $\mathit{vars}(G) \cap \mathit{vars}(\theta(G)) = \emptyset$.

Example 7.2 When examining the evaluation shown in Section 7.2.3 the altered cases of the semantic function above change the evaluation in one step. After the **Simplify** step in the evaluation for s, the p in the store is observed, so the new step is

$$\begin{array}{ll}\mathcal{S}[\![\mathcal{P}]\!](\langle s\#3 : 1, [p\#1 : 4], \{p\#1, q\#2, s\#3\}, \mathit{true}, \bullet\rangle_4) & (\textbf{Simplify}) \\ \quad = \quad \mathcal{S}[\![\mathcal{P}]\!](\langle \Box, [p\#1 : 4^*], \{q\#2\}, \mathit{true}, \bullet\rangle_4) & \end{array}$$

During the other **Simplify** and **Propagate** steps, either no rule is fired or the fired rule is single-headed and hence no constraints are observed.

7.4.2 Abstract Domain

The domain of abstract execution states used for this analysis is rather simple. We abstract CHR constraints by their predicate names, and built-in constraints

as simply the special predicate name `builtin`. The abstract state simple holds an abstraction of the goal or active occurrenced constraint, and an abstraction of the call stack A. The abstracted call stack is a set. It denotes the predicate occurrences which have not been observed.

Let c be a built-in constraint and p a CHR constraint, and S a set or multiset of CHR constraints. We define the late storage abstraction α_{ls} as follows:

$$\begin{array}{rcll}
\alpha_{ls}(c) & = & \texttt{builtin} & (c \text{ built-in}) \\
\alpha_{ls}(p(t_1,\ldots,t_n)) & = & p & \\
\alpha_{ls}(p(t_1,\ldots,t_n)\#i) & = & p & \\
\alpha_{ls}(p(t_1,\ldots,t_n)\#i:j) & = & p:j & \\
\alpha_{ls}([]) & = & [] & \\
\alpha_{ls}([c|G]) & = & [\alpha_{ls}(c)|\alpha_{ls}(G)] & \\
\alpha_{ls}(S) & = & \{\alpha_{ls}(c)|c \in S\} & (S \text{ set}) \\
\alpha_{ls}(\langle G, A, _, _, _\rangle__) & = & \langle \alpha_{ls}(G), \alpha_{ls}(\mathsf{unobserved}(A))\rangle &
\end{array}$$

where unobserved is defined as

$$\mathsf{unobserved}(A) = \left\{ p \;\middle|\; \begin{array}{l} p(t_1,\ldots,t_n)\#i:j \in \mathsf{list2set}(A), \\ \neg\exists p(t'_1,\ldots,t'_n)\#i':j'^{*} \in \mathsf{list2set}(A) \end{array} \right\}$$

$$\begin{array}{rcl}
\mathsf{list2set}([]) & = & [] \\
\mathsf{list2set}([a|A]) & = & \{a\} \cup \mathsf{list2set}(A)
\end{array}$$

Note we abstract built-in constraints, and non-identified CHR constraints by keeping the predicate. We abstract identified CHR constraints by removing the identity number and occurrenced identified CHR constraints just keeping track of the occurrence number. We eliminate observed constraints from the execution stack using the auxiliary function unobserved.

The abstracted call stack is a set. It denotes the predicates which have not been observed.

Note that program point information can easily be derived from the abstract state:

$$\mathsf{pp}(\langle G, A\rangle) = G$$

Hence, the partial ordering and least upper bound operator are only defined for abstract states with the same abstract goal: The partial ordering on states is $\langle G, A\rangle \preceq_{ls} \langle G', A'\rangle$ iff $G = G'$ and $A' \subseteq A$.

For the sake of completeness, we add a top element $\top_{ls}$ to the abstract domain, with $\gamma(\top_{ls}) = \Sigma$ and $\forall s \in \Sigma_a : s \preceq_{ls} \top_{ls}$. The value $\mathsf{pp}(\top_{ls})$ is not defined, but rather $\top_{ls}$ corresponds to all program points at once. Our analysis never produces $\top_{ls}$. If it would, that would mean that the analysis gives up and yields *no* information at all.

Clearly the abstract domain forms a lattice with the ordering relation $\preceq_{ls}$. The least upper bound operator $\sqcup_{ls}$ can be defined as follows:

$$
\begin{array}{l}
s_1 \sqcup_{ls} s_2 = \\
\quad \text{if } s_1 = \langle G, A_1 \rangle \wedge s_2 = \langle G, A_2 \rangle \\
\qquad \text{then } \langle G, (A_1 \cap A_2) \rangle \\
\qquad \text{else } \top_{ls}
\end{array}
$$

7.4.3 Abstract Semantic Function

The abstract semantic function $\mathcal{AS}$ for the late storage domain is defined below.

AbstractSolve

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \texttt{builtin}, A \rangle_n) = \langle \square, \emptyset \rangle$$

A built-in constraint may possibly trigger any constraint in the constraint store. Hence all the constraints in the call stack are *possibly observed*.

For every constraint name c, the following subcomputation needs to be run to cover all execution paths, despite the fact that no information is carried over: $\mathcal{AS}[\![\mathcal{P}]\!](\langle c, \emptyset \rangle) = \langle \square, \emptyset \rangle$.

Technically, the output state of one triggered constraint should become the input state of the next according to ω_c. Moreover, the constraints could be run in any order. However, this computation is a safe approximation, since every initial and final state has a known empty A.

Abstract(Re)Activate

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c, A \rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle c : 1, A \rangle)$$

where c is a CHR constraint.

AbstractDrop

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c : j, A \rangle) = \langle \square, A \rangle_n$$

Applicable if no occurrence j exists for CHR constraint c in $\mathcal{P}$.

AbstractGoal

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle [c_{k_1}, \ldots, c_{k_n}], A \rangle) = \langle \square, A' \rangle_n$$

where

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c_{k_i}, A \rangle) = \langle \square, A_i \rangle$$

and $A' = \bigcap_{i=1}^{n} A_i$

Technically, the output state of one goal should become the input state of the next according to the concrete refined denotational semantics. However, this definition here captures the meaning of *possibly observed* too: If a constraint in the call stack is possibly observed by any goal in a conjunction, it is possibly observed by the entire conjunction.

AbstractSimplify
Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H'_1 \setminus H'_2, d_{[j]}, H'_3 \iff g \mid C$$

then

$$\begin{array}{l} \mathcal{AS}[\![\mathcal{P}]\!](\langle c : j, A_0 \rangle) = \\ \quad \text{if } \textit{unconditional-simplify} \\ \quad\quad \text{then} \quad s_1 \\ \quad\quad \text{else} \quad s_1 \sqcup_{ls} s_2 \end{array}$$

where

$$\left\{ \begin{array}{rcl} s_1 & = & \mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{ls}(C), A_1 \rangle) \\ s_2 & = & \mathcal{AS}[\![\mathcal{P}]\!](\langle c : j+1, A_0 \rangle) \\ A_1 & = & A_0 \setminus \alpha_{ls}(H'_1 \cup H'_2 \cup H'_3) \end{array} \right.$$

The condition *unconditional-simplify* holds if r is an unconditional simplification rule, i.e. of the form $c(\bar{x}) \Leftrightarrow C$ with all $x \in \bar{x}$ distinct variables. Namely, the rule application only fails when the active constraint is not in the constraint store, this leads to a state $\langle \Box, A_0 \rangle$ which when lubbed with s_1 gives s_1 because only more constraint may become possibly observed and not less. The abstract execution state A_1 marks the partner constraints of c as possibly observed.

Otherwise (the rule is not an unconditional simplification rule) the rule application either succeeds and observes constraints, or execution proceeds with the next occurrence.

AbstractPropagate
Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H'_1, d_{[j]}, H'_2 \setminus H'_3 \iff g \mid C$$

then

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c : j, A_0 \rangle_n) = \mathcal{AS}[\![\mathcal{P}]\!](\langle c : j+1, A_4 \rangle)$$

where

$$\left\{ \begin{array}{rcl} A_1 & = & A_0 \setminus \alpha_{ls}(H_1 \cup H_2 \cup H_3) \\ A_2 & = & A_1 \cup \{c\} \\ \langle \Box, A_3 \rangle & = & \mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{ls}(C), A_2 \rangle) \\ A_4 & = & A_3 \setminus (\{c\} \setminus A_1) \end{array} \right.$$

The abstract execution stack A_1 takes into account the lookup of the partner constraints: they are observed now. In A_2 the active constraint c has been pushed onto the abstract execution stack for the execution of the body of the rule. A_3 is the resulting abstract execution stack after the execution of the body. In A_4 the constraint c is removed again from the execution stack (if some copy of c was not already present prior to A_2).

Note that the active constraint c may have been observed in the execution of C iff $c \notin A_3$. Also note that here we treat the rule as if it always could have fired. This is clearly safe.

7.4.4 Example Analysis

Consider the execution of the goal `p` with respect to the following (numbered) CHR program

```
p[1]  ==> r.
p[2]  ==> s.
p[3], s[1] <=> true.
```

The evaluation of the abstract semantic function is shown below.

$$
\begin{array}{lll}
\mathcal{AS}[\![\mathcal{P}]\!](\langle p, \emptyset\rangle) & & \text{(AbstractActivate)} \\
\quad = & \mathcal{AS}[\![\mathcal{P}]\!](\langle p:1, \emptyset\rangle) & \text{(AbstractPropagate)} \\
\quad = & \mathcal{AS}[\![\mathcal{P}]\!](\langle p:2, \emptyset\rangle) & \text{(AbstractPropagate)} \\
\quad = & \mathcal{AS}[\![\mathcal{P}]\!](\langle p:3, \emptyset\rangle) & \text{(AbstractSimplify)} \\
\quad = & \mathcal{AS}[\![\mathcal{P}]\!](\langle \Box, \emptyset\rangle) \sqcup_{ls} \mathcal{AS}[\![\mathcal{P}]\!](\langle p:4, \emptyset\rangle) & \text{(AbstractGoal,AbstractDrop)} \\
\quad = & \langle \Box, \emptyset\rangle \sqcup_{ls} \langle \Box, \emptyset\rangle & \\
\quad = & \langle \Box, \emptyset\rangle &
\end{array}
$$

For the first abstract propagation step above, the result of the abstract execution of the first rule's body is used:

$$
\begin{array}{lll}
\mathcal{AS}[\![\mathcal{P}]\!](\langle r, \{p\}\rangle) & & \text{(AbstractActivate)} \\
\quad = & \mathcal{AS}[\![\mathcal{P}]\!](\langle r:1, \{p\}\rangle) & \text{(AbstractDrop)} \\
\quad = & \langle \Box, \{p\}\rangle &
\end{array}
$$

and $A_1 = A_0 = \emptyset$, $A_2 = A_1 \cup \{p\} = \{p\}$ and $A_4 = A_3 \setminus (\{p\} \setminus A_1) = \{p\} \setminus (\{p\} \setminus \emptyset) = \emptyset$. For the second abstract propagation step in the main computation, the result of the abstract execution of the second rule's body is used:

$$
\begin{array}{lll}
\mathcal{AS}[\![\mathcal{P}]\!](\langle s, \{p\}\rangle) & & \text{(AbstractActivate)} \\
\quad = & \mathcal{AS}[\![\mathcal{P}]\!](\langle s:1, \{p\}\rangle) & \text{(AbstractSimplify)} \\
\quad = & \mathcal{AS}[\![\mathcal{P}]\!](\langle \Box, \emptyset\rangle) \sqcup_{ls} \mathcal{AS}[\![\mathcal{P}]\!](\langle s:2, \{p\}\rangle) & \text{(AbstractGoal,AbstractDrop)} \\
\quad = & \langle \Box, \emptyset\rangle \sqcup_{ls} \langle \Box, \{p\}\rangle & \\
\quad = & \langle \Box, \emptyset\rangle &
\end{array}
$$

and $A_1 = A_0 = \emptyset$, $A_2 = A_1 \cup \{p\} = \{p\}$ and $A_4 = A_3 \setminus (\{p\} \setminus A_1) = \emptyset \setminus (\{p\} \setminus \emptyset) = \emptyset$. Note that p is only possibly observed in the this last evaluation of s. Hence we can safely delay storage of p until just before the execution of the second rule's body.

7.5 Groundness analysis

In this section we illustrate the use of the abstract interpretation framework by lifting the classical groundness analysis for Prolog to CHR.

In the groundness analysis for CHR we capture the groundness of variables in the scope of rules and arguments of constraints. Variables that only occur in the constraint stores are not tracked.

Unlike typical analyses for Prolog we do not go as far as capturing groundness relations between all variables.

Sections 7.5.1 and 7.5.2 present the abstract domain and the abstract semantic function respectively. The analysis is illustrated by means of an example in Section 7.5.3.

7.5.1 Abstract Domain

While abstracting groundness properties of a CHR execution we will be interested in three parts of the concrete state, the goal, the CHR constraint store, and the built-in constraint store.

Groundness is not directly affected by CHR constraints, but only through built-in constraints of the underlying constraint domain $\mathcal{D}$. Hence, we assume that we have an abstract domain $\mathcal{G}$ for tracking groundness of the underlying constraint domain $\mathcal{D}$, providing the following:

- the operations $\alpha_{\mathcal{G}}, \preceq_{\mathcal{G}}, \sqcup_{\mathcal{G}}, \ldots$
- the abstract conjunction, denoted by $\wedge_{\mathcal{G}}$ joins two abstract descriptions
- the function $\mathsf{Aadd}_{\mathcal{G}}$ joins an abstract description with a concrete constraint
- the function $\mathsf{grounds}_{\mathcal{G}}(D)$, which returns the set of variables grounded by abstract description D
- the abstract projection function $\bar{\exists}^{\mathcal{G}}_V F$ which abstracts the projection $\bar{\exists}_V F$ the projection of F onto the variables V.

We abstract the state to an abstract goal, an abstract CHR store and an abstract built-in store. The abstract goal only removes occurrence numbers. The abstract CHR store stores for each CHR constraint the least upper bound of the underlying domain's groundness descriptions of the CHR constraint instances in the store. The abstract underlying store is an element of the domain $\mathcal{G}$ that is restricted to the variables in the goal.

$$\begin{aligned}
\alpha_g(c) &= c \ (c \text{ is built-in}) \\
\alpha_g(p(t_1,\ldots,t_n)) &= p(t_1,\ldots,t_n) \\
\alpha_g(p(t_1,\ldots,t_n)\#i) &= p(t_1,\ldots,t_n) \\
\alpha_g(p(t_1,\ldots,t_n)\#i:j) &= p(t_1,\ldots,t_n):j
\end{aligned}$$

$$\begin{array}{rcl}\alpha_g([]) & = & [] \\ \alpha_g([c|G]) & = & [\alpha_g(c)|\alpha_g(G)] \\ \alpha_g(S) & = & \{\alpha_g(c)|c \in S\} \quad (S \text{ set or multiset})\end{array}$$

$$\begin{array}{l}\alpha_g(p(t_1,\ldots,t_n)\#i, B) = p(x_1,\ldots,x_n) \leftarrow D \\ \quad \text{where } D = \bar{\exists}^{\mathcal{G}}_{x_1,\ldots,x_n} \alpha_{\mathcal{G}}(B \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n)\end{array}$$

$$\begin{array}{l}\alpha_g(S, B) = snf(\{\alpha_g(c, B)|c \in S\}) \quad (S \text{ set or multiset}) \\ \alpha_g(\langle G, \_, S, B, \_\rangle_\_) = \langle \alpha_g(G), \alpha_g(S, B), \bar{\exists}^{\mathcal{G}}_{vars(G)} \alpha_{\mathcal{G}}(B)\rangle\end{array}$$

where the function *snf* creates a normal form of the groundness description of the CHR constraint store, by ensuring there is at most one entry for every CHR predicate. It is defined as follows:

$$\begin{array}{rcl}snf(\emptyset) & = & \emptyset \\ snf(\{p(\bar{x}) \leftarrow D_1\} \uplus S) & = & snf(\{p(\bar{x}) \leftarrow D_1 \sqcup_{\mathcal{G}} D_2\} \uplus S') \\ & & \text{where } S = \{p(\bar{x}) \leftarrow D_2\} \uplus S' \\ snf(\{p(\bar{x}) \leftarrow D_1\} \uplus S) & = & \{p(\bar{x}) \leftarrow D_1\} \uplus snf(S), \\ & & \text{where } \neg\exists p(\bar{x}) \leftarrow D_2 \in S\end{array}$$

We define pred as follows:

$$\begin{array}{rcl}\mathsf{pred}(p(x_1,\ldots,x_n)) & = & p \\ \mathsf{pred}(p(x_1,\ldots,x_n) \leftarrow D) & = & p \\ \mathsf{pred}([]) & = & [] \\ \mathsf{pred}([c|G]) & = & [\mathsf{pred}(c)|\mathsf{pred}(G)]\end{array}$$

The partial ordering $\preceq_g$ on states is

$$\begin{array}{c}\langle G, S, B\rangle \preceq_g \langle G', S', B'\rangle \\ \Leftrightarrow \\ G \sim G' \wedge \exists\theta : \theta(G') \equiv G \wedge B \preceq_{\mathcal{G}} \theta(B') \wedge \\ (\forall p(\bar{x}) \leftarrow D \in S : \exists p(\bar{x}) \leftarrow D' \in \theta(S') : D \preceq_{\mathcal{G}} D')\end{array}$$

where θ is a substitution.

For the sake of completeness, we add a top element $\top_g$ to the abstract domain, with $\gamma(\top_g) = \Sigma$ and $\forall s \in \Sigma_a : s \preceq_g \top_g$. The value $\mathsf{pp}(\top_g)$ is not defined, but rather $\top_g$ corresponds to all program points at once. Similarly as for the late storage analysis, the groundness analysis never produces $\top_g$.

It is possible to verify that the abstract domain forms a lattice with the ordering relation It follows form the definition of partial ordering that all variants of the same abstract execution state are considered equal. $\preceq_g$.

The least upper bound operator $\sqcup_g$ can be defined as follows:

$$\begin{array}{l}s_1 \sqcup_g s_2 = \\ \quad \text{if } s_1 = \langle G, S, B\rangle \wedge s_2 = \langle G', S', B'\rangle \wedge G \sim G' \wedge \exists\theta : G \equiv \theta(G') \\ \quad\quad \text{then } \langle G, snf(S \cup \theta(S')), B \sqcup_{\mathcal{G}} \theta(B')\rangle \\ \quad\quad \text{else } \top_g\end{array}$$

where θ is a substitution.

7.5.2 Abstract Semantic Function

The abstract semantic function $\mathcal{AS}$ for the groundness domain is defined below.

AbstractSolve

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c, S_a \uplus S_b, B\rangle) = \langle \Box, S_k, \mathsf{Aadd}_{\mathcal{G}}(c, B)\rangle$$

Applicable when c is a built-in constraint. Define $S_a = \{p(\bar{x}) \leftarrow D \mid \bar{x} \subseteq \mathsf{grounds}_{\mathcal{G}}(D)\}$ and $S_b = \{p_i(\bar{x}_i) \leftarrow D_i \mid 1 \leq i \leq n\}$.

Let

$$\begin{array}{rcll} S_0 & = & S_a \uplus S_b & \\ s_j & = & \langle c, S_a \uplus S_b, B\rangle & , j = 0 \\ s_j & = & \langle \Box, S_j, \_\rangle = \bigsqcup_g \{s_j^i \mid s_j^i = \mathcal{AS}[\![\mathcal{P}]\!](\langle p_i(\bar{x}_i), S_{j-1}, D_i\rangle) \wedge 1 \leq i \leq n\}, & j \geq 1 \end{array}$$

and be k the smallest positive integer such that $s_k = s_{k-1}$.

Abstract(Re)Activate

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c, S, B\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle c : 1, snf(\{\alpha_g(c, B)\} \cup S), B\rangle)$$

where c is a CHR constraint.

AbstractDrop

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c : j, S, B\rangle) = \langle \Box, S, B\rangle_n$$

where no occurrence j exists for CHR constraint c in $\mathcal{P}$.

AbstractGoal

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle [c|G], S_0, B_0\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle G, S, B_0 \wedge_{\mathcal{G}} B_2\rangle)$$

where $B_1 = \bar{\exists}_{vars(c)} B_0$ and

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c, S_0, B_1\rangle) = \langle \Box, S, B_2\rangle$$

AbstractSimplify

Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H_1' \setminus H_2', d_{[j]}, H_3' \iff g \mid C$$

then

$$\begin{array}{l}\mathcal{AS}[\![\mathcal{P}]\!](\langle c:j,S,B\rangle) = \\ \quad \text{if } \textit{unconditional-simplify} \\ \quad\quad \text{then} \quad s_1 \sqcup_g s_2 \\ \quad\quad \text{else} \quad s_1 \sqcup_g s_3\end{array}$$

where there exists a θ such that $c = \theta(d_j)$, $H_1 \cup H_2 \cup H_3 \subseteq S$ and $\mathsf{pred}(H_i) = \mathsf{pred}(H_i'), 1 \leq i \leq 3$.

Suppose

$$\begin{array}{rcl} H_i & = & [p_{i1}(\bar{x}_{i1}) \leftarrow D_{i1}, \ldots, p_{in_i}(\bar{x}_{in_i}) \leftarrow D_{in_i}] \\ \theta(H_i') & = & [p_{i1}(\bar{t}_{i1}), \ldots, p_{in_i}(\bar{t}_{in_i})] \end{array}$$

Let

$$\begin{array}{rcl} D_i & = & \mathsf{Aadd}(\wedge_{\mathcal{G}}\{D_{ij} \mid 1 \leq j \leq n_i\}, \wedge_{j=1}^{n_i}(\bar{x}_j = \bar{t}_j)) \\ D & = & \bar{\exists}_{vars(\theta(C))}\mathsf{Aadd}((D_1 \wedge_{\mathcal{G}} D_2 \wedge_{\mathcal{G}} D_3 \wedge_{\mathcal{G}} B), g) \end{array}$$

Suppose that

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \theta(C), S, D\rangle) = \langle \Box, S', B'\rangle$$

Then

$$\left\{\begin{array}{rcl} s_1 & = & \langle \Box, S', B \wedge_{\mathcal{G}} (\bar{\exists}_{vars(c)} B'\rangle \\ s_2 & = & \langle \Box, S, B\rangle \\ s_3 & = & \mathcal{AS}[\![\mathcal{P}]\!](\langle c : j+1, S, B\rangle) \end{array}\right.$$

The condition *unconditional-simplify* holds if r is an unconditional simplification rule, i.e. of the form $c(\bar{x}) \Leftrightarrow C$ with all $x \in \bar{x}$ distinct variables. The abstract state s_1 is the least upper bound of the unconditional application of the simplification rule and s_2 is the result if the active constraint has already been deleted. Otherwise, either the simplification is applied (s_1) or the next evaluation proceeds with the next occurrence (s_3).

We find a possible match for each CHR constraint in the rule, assume that the guard holds, and determine the abstract underlying constraint store that must exist for the body of the rule from the matching. We execute the body of the rule with this store, without removing any constraints from the store (since we are not sure how many copies there are). The resulting abstract underlying store is projected back onto the active constraint and then added to the current store.

AbstractPropagate

Let d be the j^{th} occurrence of c in a (renamed apart) rule $r \in \mathcal{P}$:

$$r \ @ \ H_1', d_{[j]}, H_2' \setminus H_3' \iff g \mid C$$

then

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c : j, S, B\rangle) = s_1 \sqcup_g s_2$$

where there exists a θ such that $c = \theta(d_j)$, $H_1 \cup H_2 \cup H_3 \subseteq S$ and $\mathsf{pred}(H_i) = \mathsf{pred}(H'_i), 1 \leq i \leq 3$.

Suppose

$$\begin{array}{rcl} H_i & = & [p_{i1}(\bar{x}_{i1}) \leftarrow D_{i1}, \ldots, p_{in_i}(\bar{x}_{in_i}) \leftarrow D_{in_i}] \\ \theta(H'_i) & = & [p_{i1}(\bar{t}_{i1}), \ldots, p_{in_i}(\bar{t}_{in_i})] \end{array}$$

Let

$$\begin{array}{rcl} D_i & = & \mathsf{Aadd}(\wedge_{\mathcal{G}}\{D_{ij} \mid 1 \leq j \leq n_i\}, \wedge_{j=1}^{n_i}(\bar{x}_j = \bar{t}_j)) \\ D & = & \bar{\exists}_{vars(\theta(C))}\mathsf{Aadd}((D_1 \wedge_{\mathcal{G}} D_2 \wedge_{\mathcal{G}} D_3 \wedge_{\mathcal{G}} B), g) \end{array}$$

Suppose that

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle\theta(C), S, D\rangle) = \langle\Box, S', B'\rangle$$

Then

$$s_1 = \mathcal{AS}[\![\mathcal{P}]\!](\langle c : j+1, S', B \wedge_{\mathcal{G}} (\bar{\exists}_c B')\rangle)$$

is the result assuming the rule fired and

$$s_2 = \mathcal{AS}[\![\mathcal{P}]\!](\langle c : j+1, S, B\rangle)$$

is the result if the rule did not fire.

7.5.3 Example Analysis

In this example analysis we will use the following simple abstract domain $\mathcal{G}$:

- $\alpha_{\mathcal{G}}(c) = \{x | x \in vars(c) \wedge c \rightarrow \mathtt{ground}(x)\}$
- $D_1 \preceq_{\mathcal{G}} D_2 \Leftrightarrow D_1 \supseteq D_2$
- $D_1 \sqcup_{\mathcal{G}} D_2 = D_1 \cap D_2$
- $D_1 \wedge_{\mathcal{G}} D_2 = D_1 \cup D_2$
- $\mathsf{Aadd}_{\mathcal{G}}(D, c) = D \cup \{x \in vars(c) | \exists D' \subseteq D : (\forall y \in D' : \mathtt{ground}(y)) \wedge c \rightarrow \mathtt{ground}(x)\}$
- $\mathsf{grounds}_{\mathcal{G}}(D) = D$
- $\bar{\exists}_V^{\mathcal{G}} D = D \cap V$

The example program we will analyze is `primes`, see (Schrijvers 2005), extended with an appropriate `main/0` constraint:

```
main[1] <=> N = 10, candidate(N).
candidate(N)[1] <=> N = 1 | true.
candidate(N)[2] <=> prime(N), M is N - 1, candidate(M).
prime(Y)[2] \ prime(X)[1] <=> 0 =:= X mod Y | true.
```

It computes the prime numbers between 1 and 10. The abstract derivation steps for the groundness analysis of this program are the following.

For brevity the abstract stores are shown separately:

$$\begin{aligned} S_1 &= \{main\texttt{:-}\emptyset\} \\ S_2 &= S_1 \cup \{candidate(N)\texttt{:-}\{N\}\} \\ S_3 &= S_2 \cup \{prime(N)\texttt{:-}\{N\}\} \end{aligned}$$

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle main, \emptyset, \emptyset\rangle) && \text{(AbstractActivate)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle main : 1, S_1, \emptyset\rangle) && \text{(AbstractSimplify)} \\ &\quad = \langle \square, S_3, \emptyset\rangle \sqcup_g \langle \square, S_1, \emptyset\rangle \\ &\quad = \langle \square, S_3, \emptyset\rangle \end{aligned}$$

In order to obtain the above abstract simplification result, the following evaluation of the first rule's body is needed:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle [X = 10, candidate(X)], S_1, \emptyset\rangle) && \text{(AbstractGoal)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [candidate(X)], S_1, \{X\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \langle \square, S_3, \{X\}\rangle \end{aligned}$$

For the first abstract goal step, this auxiliary result is used:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle X = 10, \emptyset, S_1\rangle) && \text{(AbstractSolve)} \\ &\quad = \langle \square, S_1, \{X\}\rangle \end{aligned}$$

For the second abstract goal step, this auxiliary result is used:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X), S_1, \{X\}\rangle) && \text{(AbstractActivate)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X) : 1, S_2, \{X\}\rangle) && \text{(AbstractSimplify)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \square, S_2, \{X\}\rangle) \sqcup_g \mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X) : 2, S_2, \{X\}\rangle) \\ &\quad = \langle \square, S_2, \{X\}\rangle \sqcup_g \langle \square, S_3, \{X\}\rangle \\ &\quad = \langle \square, S_3, \{X\}\rangle \end{aligned}$$

which uses the result:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X) : 2, S_2, \{X\}\rangle) && \text{(AbstractSimplify)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [prime(X), Y \text{ is } X - 1, candidate(Y)], S_2, \{X\}\rangle) \sqcup_g \langle \square, S_2, \{X\}\rangle) \\ &\quad = \langle \square, S_3, \{X\}\rangle \sqcup_g \langle \square, S_2, \{X\}\rangle \\ &\quad = \langle \square, S_3, \{X\}\rangle \end{aligned}$$

This in turn uses:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle [prime(X), Y \text{ is } X - 1, candidate(Y)], S_2, \{X\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [Y \text{ is } X - 1, candidate(Y)], S_3, \{X\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [candidate(Y)], S_3, \{X, Y\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \langle \square, S_3, \{X, Y\}\rangle \end{aligned}$$

The evaluation for the `prime(X)` goal is as follows:

$$
\begin{array}{lll}
\mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N), S_2, \{N\}\rangle) & & \text{(AbstractActivate)} \\
\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N) : 1, S_3, \{N\}\rangle) & & \text{(AbstractSimplify)} \\
\quad = \langle \square, S_3, \{N\}\rangle \sqcup_g \mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N) : 2, S_3, \{N\}\rangle) & & \text{(AbstractPropagate)} \\
\quad = \langle \square, S_3, \{N\}\rangle \sqcup_g \mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N) : 3, S_3, \{N\}\rangle) & & \text{(AbstractDrop)} \\
\quad = \langle \square, S_3, \{N\}\rangle \sqcup_g \langle \square, S_3, \{N\}\rangle) & & \\
\quad = \langle \square, S_3, \{N\}\rangle & &
\end{array}
$$

We omit identical evaluations for $[prime(N), M\ is\ N-1, candidate(M)]$ and $prime(N)$ starting with CHR store S_3 rather than S_2. From this analysis we can conclude that the CHR constraints are ground at all times in this program.

7.6 Implementation and Evaluation

We have implemented both the late storage analysis and the groundness analysis in our K.U.Leuven CHR system (see Chapter 6).

We have implemented the late storage and groundness analyses to always start from an initial goal $\langle \texttt{main}, \emptyset\rangle$ and $\langle \texttt{main}, \emptyset, \emptyset\rangle$ respectively. The rules for the constraint `main/0` in a particular benchmark define all relevant call patterns for that benchmark.

7.6.1 Late Storage Analysis

The results of this analysis are used for optimization in our CHR compiler in the following way:

- The main philosophy in late storage is to delay constraint storage, so that some constraints are removed before they have to be stored. For those constraints the overhead of both storage and removal is then avoided.

  The reference CHR implementation in SICStus (Intelligent Systems Laboratory 2003) already has an approximate late storage optimization. Namely, it does not store an activated constraint straight away, but only ensures it is stored before a rule body of a propagation occurrence is executed. See Section 5.3 for a more extensive treatment of late storage.

  With our late storage analysis, the optimization of Section 5.3 is made stronger: our compiler now also avoids the storage of an active constraint before the execution of a body of a propagation occurrence, if the constraint is not observed during the execution of that body.

- For a particular class of constraints, our compiler derives that they are *never stored*. Never stored constraints are not stored before an unconditional simplification occurrence. An unconditional simplification occurrence, is an oc-

currence in a single-headed rule without any matching or guard. The following optimizations are possible for never stored constraints:

- A constraint that is never stored, cannot be triggered. Hence no checks are necessary to distinguish between activation and reactivation.
- A never stored constraint cannot be found in a constraint store. Hence if it occurs in a multi-headed rule, its partner constraints in that rule should not actively try to apply that rule, i.e. their occurrences are considered passive.
- A never stored constraint will not reconsider the same propagation rule twice with the same partner constraints. Hence no history needs to be maintained for that rule.

Hence, the code generated by our compiler is much closer to the code one would write for a deterministic procedure in the host language than for an arbitrary constraint without the never stored property.

In Table 7.2 we show the speed-ups resulting from late storage analysis in hProlog. For eight benchmarks, see (Schrijvers 2005), we compare immediate storage with the current implementation of the above optimizations that are enabled by late storage analysis. The timings for the optimized programs are given relative to those of the unoptimized programs.

Benchmark	Optimized / Unoptimized
bool	17.6%
fib	72.3%
fibonacci	72.7%
leq	75.7%
mergesort	86.5%
primes	94.6%
uf	97.4%
uf_opt	106.5%
wfs	95.7%
zebra	89.1%

Table 7.2: Late storage analysis: runtime results of optimized programs relative to unoptimized programs

In Table 7.3 we show the number of dynamic constraint store insertions and deletions for these benchmarks.[1] The considerable reduction of the `bool` benchmark timing is clearly explained by the drastic decrease in the number of store

[1]Note that in the zebra benchmark the number of deletions is larger than the number of additions. This is due to backtracking over deletions.

operations. While even more operations have been saved in the `leq` benchmark, the impact on its runtime is more modest, though still considerable. Measurement indicates that the impact of these operations on the total runtime is less dominant and so less overall improvement can be realized.

Benchmark	Without		With	
	Insert	Delete	Insert	Delete
bool	359,996	359,996	8.33%	8.33%
fib	114,603	114,580	50.01%	50.00%
fibonacci	81,000	39,000	51.85%	0.00%
leq	34,280	34,280	5.16%	5.16%
mergesort	37,170	34,610	30.97%	25.86%
primes	4,999	4,632	49.99%	46.03%
wfsnew	46,800	44,800	91.03%	90.62%
uf	7,994	6,994	37.50%	28.57%
uf_opt	8,004	7,004	37.50%	28.57%
zebra	56,790	130,300	37.52%	28.60%

Table 7.3: Late storage analysis: the number of store operations without and with late storage

The analysis time for the late storage analysis is reasonable, in the range of 0 tot 20 ms for the above benchmarks and mostly only a fraction of total compilation time. For the K.U.Leuven CHR system compiler, including 76 constraints and 144 CHR rules, the analysis takes 500 ms or a fifth of total compilation time.

7.6.2 Groundness Analysis

Our implementation of groundness analysis uses the naive groundness domain for built-in constraints as it is described in Section 7.5.3.

The K.U.Leuven CHR system currently only performs optimizations for constraints that are ground in all possible states. The optimizations are enabled by groundness declarations that are supplied by the programmer (see Section 6.3). In order to evaluate our analysis we have used the results of the analysis to automatically infer the groundness declarations.

We have experimentally evaluated our groundness analysis in this way on the seven benchmarks also used in Section 6.5.3: `fib`, `fibonacci`, `mergesort`, `primes`, `uf`, `uf_opt` and `wfs`. To each of these benchmarks we have added a `main/0` constraint representative of the use of the particular benchmark.

It turns out that the annotations derived from the groundness analysis results are optimal for all but the union-find benchmarks. Optimal means that the derived annotations are as strong as the actual calling patterns of the constrains in those

benchmarks. The speed-ups realized by the annotations were listed in Table 6.2 in Section 6.5.3.

For the union-find benchmark, the results are not optimal. The analysis does not figure out that the first argument of `find/2` and both arguments of `link/2` are always ground, because the analysis does not take into account that a `find/2` is only called on an element that already appears in a `root/2` or `~>/2` constraint and never delays. Although the derived annotations are not optimal, the speed-ups are about as good as for the optimal annotations: for the `uf` benchmark we measured no difference and for the `uf_opt` benchmark we measured a difference of at most 10 ms.

The analysis time of the groundness analysis is more troublesome than that of the late storage analysis. Times are in the range of 10 to 100 ms for the smaller benchmarks (`fib`, `fibonacci`, `mergesort`, `primes`) with 2 to 4 CHR rules and in the range of 10,000 ms for the larger ones (`uf`, `uf_opt` and `wfs` with respectively 6, 7 and 44 CHR rules). In all cases the groundness analysis dominates the total compilation time.

7.7 Conclusion

To the best of our knowledge, this is the first work on using abstract interpretation for CHR. Many ad-hoc analyses and optimizations were developed for CHR before: delay avoidance (Holzbaur, García de la Banda, Stuckey, and Duck 2005; Schrijvers and Demoen 2004a) (see Section 6.3.4), late storage, continuation optimization, an index optimization (Holzbaur, García de la Banda, Stuckey, and Duck 2005), ... Typically the analysis process used to obtain information that enables a particular optimization is only discussed informally or left out altogether.

We have shown that it is possible to apply the general and structured ideas of abstract interpretation to CHR. Based on our definition of the refined denotational semantics of CHR, we have formulated a framework for abstract interpretation. To illustrate the framework we have formulated two analyses in it: the CHR specific late storage analysis and the groundness analysis which we have lifted from Prolog to CHR. These two domains show that it is possible to precisely and formally state program analyses for CHR which yield useful information for program optimization.

Our work on abstract interpretation has been published as a technical report (Duck, Schrijvers, and Stuckey 2004) and accepted at the Principles and Practice of Declarative Programming Symposium (Schrijvers, Stuckey, and Duck 2005).

7.7.1 Related and Future Work

The most closely related work we are aware of is the one on the analysis of concurrent constraint logic (CCL) programs (Codognet, Codognet, and Corsini 1990;

Codish, Falaschi, Marriott, and Winsborough 1993). CCL programs roughly correspond to CHR programs with only single-headed simplification rules. The CCL semantics corresponds to the high-level operational semantics of CHR. Due to the single-headedness of CCL programs, their analysis is not complicated by the non-determinism of the partner constraint matchings of CHR. However, their semantics is more non-deterministic regarding the order of rule-applications than the refined operational semantics we use.

We have only presented two rather straightforward analysis domains as an illustration of the framework. These analyses should of course be strengthened with additional control flow information that is derived from other analyses. It is for example possible to derive the never stored property for some constraints from the late storage analysis. This information reduces the set of constraints that may be reactivated.

Moreover the groundness analysis has only been exploited in the case that arguments of constraints are ground throughout their full lifetime. As an extension, one can exploit the groundness information in other cases, i.e. when arguments are ground from a certain occurrence on or at certain occurrences.

Many more analyses for CHR can be considered within the framework as well as the combination of these analyses.

Several efficiency issues have risen during the formulation of our framework, namely due to the fixedpoint computations. It remains to be explored how much the impact of these computations is on the overall efficiency of analyses in our framework. Widening strategies can be applied to avoid overly long analysis times for some domains. A comprehensive study of the time/accuracy trade-off is required.

For a specific CHR compiler the accuracy can be improved across analysis domains by using the more specialized operational semantics of the compiler. The semantics of the compiler will typically be a more deterministic instance of the denotational semantics.

The common abstract interpretation analysis technique may facilitate the more unified view of host language and CHR to perform multi-language analysis. For example in the case of CHR in Prolog, a single groundness analysis for both the Prolog code and its embedded CHR code would obtain the strongest results since there is a reciprocal interaction between both languages. A more unified semantics of both is necessary to accomplish this.

Chapter 8

Integration of CHR with Tabled Execution

8.1 Introduction

XSB (Warren et al. 2005) is a standard Prolog system extended with *tabled resolution*. Tabled resolution is useful for recursive query computation, allowing programs to terminate in many cases where Prolog does not.

Parsing, program analysis, model checking, data mining, diagnosis and many more applications benefit from tabled resolution.

The use of constraint solvers in XSB has been a quite laborious and inconvenient endeavor up to now. Initially XSB provided no built-in support at all for dealing with constraints. Hence XSB programmers resorted to interfacing with foreign language libraries or implementation of constraint solvers in XSB itself with close coupling of constraint solver and application as a consequence. For instance, the initial feasibility study of a real time model checking system used a meta interpreter written in XSB to deal with constraints (see (Mukund, Ramakrishnan, Ramakrishnan, and Verma 2000)). The subsequent full system implements an interface between XSB and the POLINE polyhedra-based constraint solver library and passes around handles to the constraint store in the XSB program (see (Du, Ramakrishnan, and Smolka 2000)). A later version of this real time model checking application switched to using distance bound matrices implemented in XSB itself (see (Pemmasani, Ramakrishnan, and Ramakrishnan 2002)). This shows that there is certainly a demand for constraints in the XSB setting, but that a satisfactory solution with sufficient ease of use and a reasonable implementation had not been found so far.

In an attempt to facilitate the use of constraints in XSB, it has been extended with attributed variables (Cui and Warren 2000a). Attributed variables form

a Prolog language feature that is particularly suited for constraint solver implementation as it allows efficient association of data with variables and user hooks on variable binding. Unfortunately this feature has not caught on in XSB as a basis for constraint systems because it is a particularly low-level feature that still requires considerable scheduling considerations by the constraint solver programmer. However, most of the work on attributed variables in XSB is still useful, as attributed variables are indeed a powerful implementation tool for constraint systems: efficient compilation of CHR to Prolog relies heavily on them (see Section 5.2.3). Because of the availability of attributed variables, CHR is a good high-level alternative for the previous low-level approaches.

From the point of view of CHR, the integration of CHR with XSB's tabled resolution yields a more powerful system than the combination of plain Prolog and CHR. The CHR-XSB integration combines both the bottom-up and top-down fixedpoint computations, the favorable termination properties of XSB and the constraint programming of CHR. This combined power enables programmers to easily write highly declarative programs that are easier to maintain and extend.

This proposed integration implies quite a number of implementation challenges that have to be dealt with. Firstly, a CHR system for non-tabled use in XSB is needed, as it does not come with one already. This has already been discussed in Section 6.4.1: we have ported the K.U.Leuven CHR system to XSB.

Secondly, the K.U.Leuven CHR system requires proper interaction with tabled execution. Section 8.2 presents the necessary technical background: SLD resolution and its extension to constraints. In Section 8.3 we present our implementation and the different interaction issues. We propose a solution to the conflict of the global CHR store with the required referential transparency of tabled predicates. Also two representations of tabled constraints are analyzed and a mechanism for answer constraint projection is presented. In addition, we study issues regarding answer entailment checking and advanced techniques for aggregate-like answer combination. The performance impact of tabling and several of the mentioned concepts are measured on a small constraint application. Throughout, we propose a declaration-based approach for enabling particular mechanisms that, in the spirit of both CHR and XSB, preserves the ease of use.

Finally, we conclude this chapter in Section 8.4 and discuss related and possible future work.

8.2 Technical Background

8.2.1 SLG Resolution

The tabled execution of XSB is based on SLG resolution. SLG resolution is an execution strategy that shares the improved termination behavior with bottom-up execution and the goal-directedness with top-down execution. It implements the

Parent	Children	Conditions
Clause Resolution		
$\texttt{root}(G)$	$\texttt{body}(G; B_1^1, \ldots, B_k^1)\theta_1$ $\vdots$ $\texttt{body}(G; B_1^l, \ldots, B_{k_l}^l)\theta_l$	for all $0 < i \leq l$ such that $G_i \rightarrow B_1^i, \ldots, B_{k_i}^i \in P$ and θ_i the mgu of G and G_i
Answer Propagation		
$\texttt{body}(G; B_1, B_2 \ldots, B_k)$	$\texttt{body}(G; B_2, \ldots, B_k)\theta_1$ $\vdots$ $\texttt{body}(G; B_2, \ldots, B_k)\theta_l$	for all $A_i \in ans(B_1)$ where θ_i is the mgu of B_1 and A_i

Table 8.1: Basic SLG resolution rules

fixedpoint semantics of the immediate consequence operator $\mathbf{T}_P$ (see Definition 3.5 on page 42).

This execution strategy is based on the rewriting rules given in Table 8.1 where G and B_i are atoms and P is a definite logic program. An SLG-tree is a tree built from a node $\texttt{root}(G)$ by a finite application of the rewriting rules. When there is no tree yet for a goal B_1 to be resolved by the *Answer Propagation* rule, a new SLG-tree in the SLG-forest is created with root $\texttt{root}(B_1)$.

The *Answer Propagation* rule implements the left-to-right selection rule implemented by XSB and common to most LP systems.

The SLG rewriting rules are used for query evaluation according to the definition below.

Definition 8.1 *Let $SLG(G)$ be the SLG-forest created from the query (G, P) as follows:*

1. *Create an SLG-forest containing a single tree $\{\texttt{root}(G)\}$.*
2. *Expand the leftmost node using the rules in Table 8.1 as long as they can be applied.*
3. *Return the set $ans(G)$ as the answer for the query.*

Let $slg(G)$ be the tree rooted by $\texttt{root}(G)$.

Definition 8.2 *Let $ans(G)$ be the set of all A such that $\texttt{body}(A; \square) \in slg(G)$.*

The set $ans(G)$ is also called the (answer) table for G. XSB maintains this answer table in a data structure and does the necessary bookkeeping to ensure that all rewriting rules are applied when appropriate.

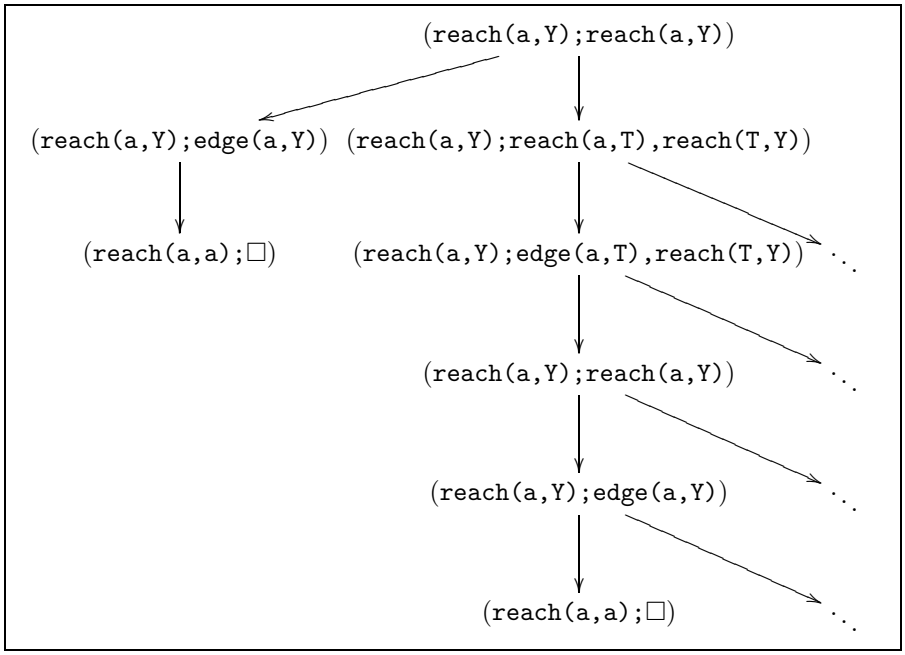


Figure 8.1: SLD-tree example

Example 8.1 Consider the following logic program:

```
edge(a,a).

reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,T), reach(T,Y).
```

The predicate `edge/2` represents the edges in a graph and the predicate `reach/2` implements graph reachability. With the given fact `edge(a,a)` the graph represented is a single node `a` with a loop.

Under the SLDNF strategy of Prolog (see Section 2.3.1) the query `?- reach(a,Y)` produces an infinite number of answers of the form `reach(a,a)` due to the loop in the graph. The SLDNF strategy implies a resolution tree that starts from the query and in each step replaces an unresolved literal with the body of a corresponding rule. The SLDNF tree for the query is depicted in Figure 8.1. We also maintain the substituted initial query throughout the tree; the leaves of the tree are the answers to the query.

Under SLG resolution only one answer is produced and execution terminates

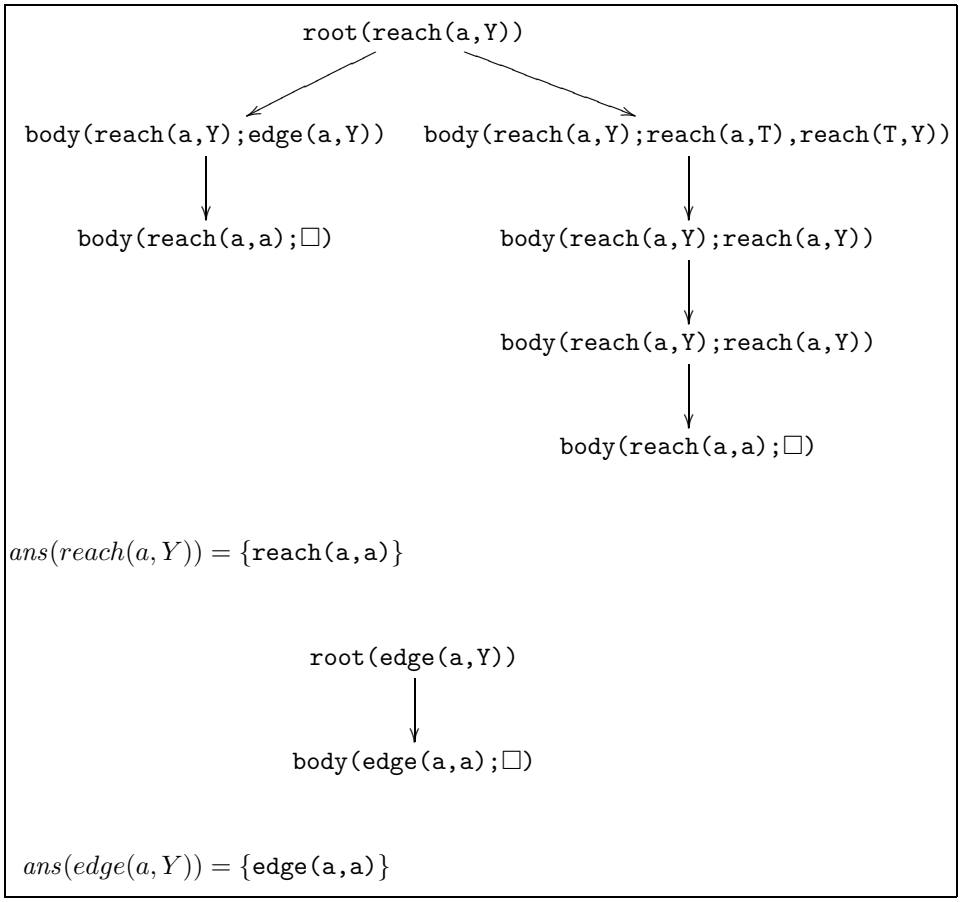


Figure 8.2: SLG-tree example

finitely, as is shown in Figure 8.2.

In (Rao, Sagonas, Swift, Warren, and Freire 1997) XSB's SLG execution strategy has been extended with support for negated literals in order to realize the well-founded semantics. See Section 3.4.1 for a definition of the well-founded semantics.

We refer the reader to (Chen and Warren 1996) for an extensive coverage of XSB's SLG execution strategy of tabled resolution.

8.2.2 SLG and Constraints: SLG$^{\mathcal{D}}$ Resolution

In a survey of Constraint Logic Programming (CLP) (Jaffar and Maher 1994) various forms of semantics are listed for constraint logic programs: declarative, fixedpoint and operational semantics. The fixedpoint semantics are of particular interest as we can relate them to XSB's fixedpoint semantics.

The CLP fixedpoint semantics are defined, in the usual way, as the fixedpoint of an extended immediate consequence operator.

Definition 8.3 (CLP Immediate Consequence Operator) *The one-step consequence function $T_P^{\mathcal{D}}$ for a CLP program P with constraint domain $\mathcal{D}$ is defined as:*

$$T_P^{\mathcal{D}}(I) = \{p(\bar{d}) \mid \begin{array}{l} p(\bar{x}) \leftarrow c, b_1, \ldots, b_n \text{ is a rule in } P \\ \forall i : 1 \leq i \leq n \Rightarrow a_i \in I \\ v \text{ is a valuation on } \mathcal{D} \text{ such that} \\ \mathcal{D} \models v(c), v(\bar{x}) = \bar{d}, \\ \forall i : 1 \leq i \leq n \Rightarrow v(b_i) = a_i\} \end{array}$$

The valuation is a value assignment or grounding of the variables.

A goal-directed execution strategy, SLG$^{\mathcal{D}}$, using tabling for the above CLP fixedpoint semantics has been developed by Toman in (Toman 1997).

As SLG$^{\mathcal{D}}$ is an extension of SLG, we shall only point out the differences. SLG$^{\mathcal{D}}$ generalizes Herbrand constraints (i.e. unifications) to a constraint domain $\mathcal{D}$. In (Toman 1997) it is assumed that the constraint domain $\mathcal{D}$ comes with a projection operation that returns a disjunction of constraints: $\bar{\exists}_T C = \bigvee_i C_i$. The notation $C_j \in \bar{\exists}_T C$ is used to state that C_j is one of the disjuncts in this disjunction. In addition, it is assumed that some relation $\leq_{\mathcal{D}}$ is provided. This relation should be stronger than implication, i.e.

$$\forall C_1, C_2 : C_1 \leq_{\mathcal{D}} C_2 \quad \Rightarrow \quad \mathcal{D} \models C_1 \rightarrow C_2$$

A query in the SLG$^{\mathcal{D}}$ formalism is a tuple (G, C, P) where $vars(C) \subseteq vars(G)$ and all arguments of G are variables. With this new definition of a query the rewriting rules become those in Table 8.2. There are two new rules: *Query Projection* and *Answer Projection*. The former is responsible for determining the goals to be resolved by answer propagation. The latter is responsible for determining the computed answers to be stored. The new definition of $ans(G, C)$ is:

Definition 8.4 *The answer set of the query (G, C, P), denoted $ans(G, C)$, is the set of all A such that $\mathtt{ans}(G; A) \in slg(G, C)$.*

In (Toman 1997) several optimizations to the rewriting formulas have been proposed, of which one to *Query Projection* will be of particular interest. The optimization allows for more general goals than strictly necessary to be resolved. Table 8.3 lists the modified *Query Projection* rule.

Parent	Children	Conditions
Clause Resolution		
$\texttt{root}(G; C)$	$\texttt{body}(G; B^1_1, \ldots, B^1_{k_1}; C \wedge \theta \wedge D^1)$ $\vdots$ $\texttt{body}(G; B^l_1, \ldots, B^l_{k_l}; C \wedge \theta \wedge D^l)$	for all $0 < i \leq l$ such that $G' \rightarrow D^i, B^i_1, \ldots, B^i_{k_i}$ and $\theta \equiv (G = G')$ and $C \wedge \theta \wedge D^i$ is satisfiable
Query Projection		
$\texttt{body}(G; B_1, \ldots, B_k; C)$	$\texttt{goal}(G; B_1, C_1; B_2 \ldots, B_k; C)$ $\vdots$ $\texttt{goal}(G; B_1, C_l; B_2 \ldots, B_k; C)$	for all $C_i \in \bar{\exists}_{B_1} C$
Answer Propagation		
$\texttt{goal}(G; B_1, C_1; B_2 \ldots, B_k; C)$	$\texttt{body}(G; B_2, \ldots, B_k; C \wedge \theta \wedge A_1)$ $\vdots$ $\texttt{body}(G; B_2, \ldots, B_k; C \wedge \theta \wedge A_l)$	for all $A_i \in ans(B', C')$ where $\theta \equiv (B' = B_1)$ and $C_1 \wedge \theta \leq_{\mathcal{D}} C'$ and $C \wedge \theta \wedge A_i$ is satisfiable
Answer Projection		
$\texttt{body}(G; \square; C)$	$\texttt{ans}(G; A_1)$ $\vdots$ $\texttt{ans}(G; A_l)$	for all $A_i \in \bar{\exists}_G C$

Table 8.2: SLG$^{\mathcal{D}}$ resolution rules

Parent	Children	Conditions
Query Projection		
$\texttt{body}(G; B_1, \ldots, B_k; C)$	$\texttt{goal}(G; B_1, C_1; B_2 \ldots, B_k; C)$ $\vdots$ $\texttt{goal}(G; B_1, C_l; B_2 \ldots, B_k; C)$	$\mathcal{D} \models \bar{\exists}_{B_1} C \rightarrow C_1 \vee \ldots \vee C_l$ for some $C_1, \ldots, C_l$

Table 8.3: Optimized *Query Projection* for SLG$^{\mathcal{D}}$ resolution

A second important optimization is a modified version of the answer set definition:

Definition 8.5 *The answer set of the query (G, C, P), denoted $ans(G, C)$, is the set of all A such that $\mathtt{ans}(G; A) \in slg(G, C)$ and no A' is already in $ans(G, C)$ for which $A \leq_{\mathcal{D}} A'$.*

This alternative definition allows for answers to be omitted if they are already entailed by earlier more general answers. While logically the same answers are entailed, the set of answers is smaller with the new definition.

In essence our integration of CHR with tabled execution is an implementation of the SLG$^{\mathcal{D}}$ execution strategy and we will point out the correspondences in Section 8.3.

In (Toman 1996) Toman has also extended his work to a goal-directed execution strategy for CLP programs with negation. This extension realizes the well-founded semantics. An implementation of this extension is not covered by our work. It imposes additional requirements on the constraint solver: a finite representation of the negation of any constraint should exist. Moreover, the detection of loops through negation requires a more complicated tabling mechanism.

8.3 CHR and Tabled Execution

The main challenge of introducing CHR in XSB is the integration of CHR constraint solvers with the backward chaining fixedpoint computation of SLG resolution according to the SLG$^{\mathcal{D}}$ semantics of the previous section.

A similar integration problem has been solved in (Cui and Warren 2000a), which describes a framework for constraint solvers written with attributed variables for XSB. The name Tabled Constraint Logic Programming (TCLP) is coined in that publication, though it is not formulated in terms of SLG$^{\mathcal{D}}$ resolution. Porting CHR to XSB was already recognized as important future work of (Cui and Warren 2000a).

The main difference for the programmer between CHR and attributed variables for developing constraint solvers, i.e. the fact that CHR is a much higher level language, should be carried over to the tabled context. Hence tabled CHR should be a more convenient paradigm for programming constraint solvers than TCLP with attributed variables. Indeed, we will show how the internals presented in this section can be hidden from the user.

In (Cui and Warren 2000a) the general TCLP framework specifies three operations to control the tabling of constraints: call abstraction, entailment checking of answers and answer projection. These operations correspond with the optimization to *Query Projection*, the projection in *Answer Projection* and the compaction

of the $ans(G;C)$ set. It is left to the constraint solver programmer to implement these operations for his particular solver.

In the following we formulate these operations in terms of CHR. The operations are covered in significant detail as the actual CHR implementation and the representation of the global CHR constraint store are taken into account. Problems that have to be solved time and again for attributed variable constraint solvers are solved once and for all for CHR constraint solvers. Hence integrating a particular CHR constraint solver requires much less knowledge of implementation intricacies and decisions can be made on a higher level.

Overview The general implementation schema is presented in Section 8.3.1: it introduces the refinements covered in the subsequent sections, Section 8.3.2 to Section 8.3.5 Finally, Section 8.3.6 uses a small shipment problem to evaluate the combination of CHR with tabling.

8.3.1 General Schema of the Implementation

We present a source-to-source transformation that maps a predicate p onto a predicate p' such that the answers to p under SLG$^{\mathcal{D}}$ and p' and under SLG are *the same*.

An SLG$^{\mathcal{D}}$ constraint goal $(p(\bar{X});C;P)$ is mapped onto an SLG goal $(p(\bar{X},C');P')$. Here C' is a term representation for the constraint C. On the level of the implementation C corresponds with the CHR constraint store and C' is a corresponding constraint store representation that is suitable for passing as an argument and storing in a table. Section 8.3.2 studies two alternative representations of C'. Note that Herbrand constraints, i.e. unifications, are still applied as most general unifiers.

For the *Query Projection* rewriting rule, the improved version of Table 8.3 is used. This fact is used:

$$\forall C : \bar{\exists}_B \rightarrow true$$

to project away all CHR constraints and replace them with `true` for all queries. In (Cui and Warren 2000a) this query projection is called call abstraction. Section 8.3.3 the motivation for our call abstraction and its realization are discussed. The conjunction of a goal's answer constraint with the current constraint store in the *Answer Propagation* rewriting rule is also covered in this section.

The projection applied in the *Answer Projection* rule is addressed in Section 8.3.4. This projection is realized as a call to a projection predicate that reduces the constraint store to its projected form.

The optimization of the Definition 8.5 is called the *entailment checking* optimization in (Cui and Warren 2000a). We use an entailment checking operation to compare a new answer with previous answers. This is discussed in Section 8.3.5.

The different steps in handling a call to a tabled predicate are depicted in Figure 8.3.

Figure 8.3: Tabled call flowchart

8.3.2 Tabled Store Representation

In this section we present two alternative constraint store representations. These are the required properties of a representation:

- The representation has to be suitable for passing it as an argument in a predicate and for storing it in an answer table.
- The representation should allow comparing the constraints in a call with all call patterns in the call table to select either a variant (under variant based tabling) or the most specific generalization (under subsumption based tabling).
- It should be possible to convert from the ordinary representation of the constraints and back, for insertion into the call table and retrieval from the answer table.

Two tabled CHR Store Representations

First we recall briefly the ordinary constraint store representation, covered in Section 5.2.2 and Section 5.2.3.

The global CHR constraint store is an updateable term, containing suspended constraints grouped by their functor. Each suspended constraint is represented as a suspension term, including the following information:

- The unique constraint identifier (ID), used for equality testing.
- The continuation goal, executed on reactivation. This goal contains the suspension itself as an argument and it is in fact a cyclic term.
- The propagation history containing for each propagation rule the tuple of identifiers of other constraints that this constraint has interacted with.

Variables involved in the suspended constraints behave as indexes into the global store: they have the suspensions attached to them as attributes.

One implicit aspect of CHR execution under the refined operational semantics is the order in which constraints are processed. Ordering information is not maintained explicitly. Without any additional support, it is not straightforward to maintain this ordering information for tabled constraints. However, in the spirit of tabling, the declarative meaning of a program rather than its operational behavior is of importance. For that reason we shall not attempt to realize the ordering of the refined operational semantics. From the user's point of view, the CHR constraints will behave according to the theoretical operational semantics and no assumptions should be made about ordering.

Two different tabled CHR store representations have been explored: the *suspension* representation and the *naive* representation. A discussion of their respective merits and weaknesses as well as an evaluation follow.

Suspension representation Here we aim to keep the tabled representation as close as possible to the ordinary representation. The idea is to maintain the propagation history of the constraints. In that way no unnecessary re-firing of propagation rules will occur after the constraints have been retrieved from the table.

However, it is not possible to just store the ordinary constraint suspensions in the table as they are. Firstly, the tables do not deal with cyclic terms. This can be dealt with by breaking the cycles before storage and resetting them after fetching. Secondly, the unique identifiers have to be replaced after fetching by fresh ones as multiple calls would otherwise create multiple copies of the same constraints all with identical identifiers. Fortunately, attributed variables themselves can be stored in tables (see (Cui and Warren 2000b)).

Naive Representation The naive representation aims at keeping the information in the table in as simple a form as possible: for each suspended constraint only the goal to impose this constraint is retained in the table. It is easy to create this goal from a suspension and easy to merge this goal back into another constraint store: it needs only to be called.

Whenever it is necessary the goal will create a suspension with a fresh unique ID and insert it into the constraint store. However it may prove unnecessary to do so. Recall the late storage optimization of Section 5.3.1. Part of this optimization consists in delaying ID creation for as long as possible in the hope that the constraint is removed before the ID is necessary. As the newly called constraint has all the constraints in the calling context to interact with, it may well be that some may serve as partner constraints to a simplification rule. If the occurrence of the newly called constraint comes before the ID creation, the ID overhead is saved out.

The only information that is lost in this representation is the propagation history. This may lead to multiple propagations for the same combination of head constraints. For this to be sound, a further restriction on the CHR rules is required: they should behave according to set semantics, i.e. the presence of multiple identical constraints should not lead to different answers modulo identical constraints.

Evaluation of both representations To measure the relative performance of the two presented representations, consider the following two programs:

prop

```
:- constraints a/1.
  a(0) <=> true.
  a(N) ==> N > 0
       | M is N - 1, a(M).

  p(N) :- a(N).
```

simp

```
:- constraints a/1, b/1.
  b(0) <=> true.
  b(N) <=> N > 0
       | a(N), M is N - 1, b(M).

  p(N) :- b(N).
```

For both programs the predicate `p(N)` puts the constraints `a(1)`...`a(N)` in the constraint store. The `prop` program uses a propagation rule to achieve this while the `simp` program uses an auxiliary constraint `b/1`. The non-tabled version of the query `p(N)` has time complexity $\mathcal{O}(N)$ for both the `simp` and the `prop` program.

In order to shield the user from the implementation details we propose this user-provided declaration through which the tabling of constraints can be controlled:

```
:- table_chr f(_,_) with Options.
```

Its meaning is that the predicate `f/2` should be tabled (using SLG$^{\mathcal{D}}$ resolution) and that it involves CHR constraints. A list of additional options may be provided. The declaration drives a source-to-source transformation that realizes the desired

Table 8.4: Evaluation of the two tabled store representations.

	`representation(suspension)`		`representation(naive)`	
program	runtime	space	runtime	space
prop	150	2,153,100	1,739	270,700
simp	109	1,829,100	89	270,700

behavior. The specifics of this transformation, the possible options and the reason for the explicit arguments in the declaration are explained later.

The two possible representations for the answer constraint store can be specified in the tabling declaration as follows:

```
:- table_chr p(_) with [representation(suspension)].
```

and

```
:- table_chr p(_) with [representation(naive)].
```

Table 8.4 gives the results for the tabled query `p(400)`: runtime in milliseconds and space usage of the tables in bytes. For both programs the answer table contains the constraint store with the 400 `a/1` constraints.

Most of the space overhead is due to the difference in representation: a suspension contains more information than a simple call. However, the difference is more or less a constant factor. The only part of a suspension in general that can have a size greater than $\mathcal{O}(1)$ is the propagation history, that for `prop` is limited to remembering that the propagation rule has been used. For the `simp` program the propagation history is always empty.

The runtime of the `prop` version with the suspension representation is considerably better than that of the version with the naive representation. In fact, there is a complexity difference. When the answer is retrieved from the table for the suspension representation, the propagation history prevents re-propagation. Hence answer retrieval is $\mathcal{O}(N)$. For the naive representation on the other hand, every constraint `a(I)` from the answer will start propagating and the complexity of answer retrieval becomes $\mathcal{O}(N^2)$.

On the other hand, for `simp` propagation history plays no role. The runtime overhead is mostly due to the additional overhead of the pre- and post-processing of the suspension representation as opposed to the simpler form of the naive representation. As a basis of comparison, in untabled versions of both programs the query takes only 10 milliseconds.

Variant Checking

The need to check whether two constraint stores are variants of each other may arise at two occasions:

- With no or only partial call abstraction (see Section 8.3.3) a constraint store is part of the call to the tabled predicate. The tabling system then needs to check whether a previous call with a variant of that constraint store appears in a table. If that is the case, the answer to the previous call can be reused.
- A limited form of entailment checking (see Section 8.3.5) consists in checking whether a new answer constraint store is a variant of any previous answer constraint store for the same call. In that case the new answer can be discarded.

Consider this equality checking with the previously presented naive tabled representation of constraints. In that representation the tabled constraints are kept as a list of goals that impose the constraints. Any permutation of this list represents the same constraint store. If two constraint stores are identical modulo variable renaming, then they are variants.

All general algorithms for variant checking of constraint stores that we are aware of have exponential worst-case complexity. A naive algorithm would be to consider all permutations of one constraint store. If any one of the permutations equals the other constraint store, both are identical. With heuristics this algorithm could be improved and for particular constraints or even applications algorithms with a better complexity may exist. However further exploration of improvements to variant checking falls outside of the scope of this text. The problem can be ignored altogether, with possible duplication in tables as a consequence, or only partially tackled, e.g. by simple sorting and pattern matching.

In Chapter 9 a technique for semantically founded implication checking is developed. If two constraint stores imply each other, they are equivalent. This approach would be stronger than the strictly syntactical approaches above in the sense that more constraint stores are decided to be variants, but it could also be much more expensive.

8.3.3 Call Abstraction

Call abstraction replaces the called goal with a call to a more general goal followed by an operation that ensures that only the answer substitutions applicable to the original call are retained. At the level of plain Prolog, abstraction means not passing certain bindings to the call. For example, `p(q,A)` can be abstracted to `p(Q,A)`. This goal has then to be followed by `Q = q` to ensure that only the appropriate bindings for `A` are retained.

In XSB call abstraction is a means to control the number of tables. When a predicate is called with many different instantiation patterns, a table is generated

for each such call instantiation pattern. Thus it is possible that the information for one fully instantiated call is present many times in tables for different call instantiation patterns. This duplication in the tables can be avoided by using call abstraction to obtain a smaller set of call instantiation patterns.

For constraint logic programming, call abstraction can be generalized from bindings to constraints: abstraction means removing some of the constraints on the arguments. Consider for example the call `p(Q,A)` with constraint `Q leq N` on `Q`. This call can be abstracted to `p(Q',A)`, followed by `Q'=Q` to reintroduce the constraint.

Abstraction is particularly useful for those constraint solvers where the number of constraints on a variable can be much larger than the number of different bindings for that variable. Consider for example a finite domain constraint solver with constraint `domain/2`, where the first argument is a variable and the second argument the set of its possible values. If the variable can be bound to at most n values it can take as many as 2^n different `domain/2` constraints, one for each subset of values. Thus many different tables would be needed to cover every possible call pattern.

Varying degrees of abstraction are possible and may depend on the particular constraint system or application. Full constraint abstraction, i.e. the removal of all constraints from the call, is generally more suitable for CHR for the following reasons:

- CHR rules do not require constraints to be on variables. They can be on ground terms or atoms as well. It is not straightforward to define abstraction for ground terms as these are not necessarily passed in as arguments but can just as well be created inside the call. Hence there is no explicit link with the call environment, while such a link is needed for call abstraction. As such, only "no abstraction" or full constraint abstraction seem suitable for CHR.

- Full constraint abstraction is preferable when the previously mentioned table blow-up is likely.

- As mentioned in the previous section, variant checking of constraint stores can have exponential complexity.

Moreover, it may be costly to sort out what constraints should be passed in to the call or abstracted away. Hence often full abstraction is cheaper than partial abstraction. For instance, consider a typical propagation-based finite domain constraint solver with binary constraints only. The constraint graph for a number of such finite domain constraints has a node for every variable involved in a constraint and an edge between variables involved in the same constraint. Any additional constraint imposed on a variable in a component of the graph may affect the domain of all other variables in the same component. A call abstraction that would restrict the constraint store to all constraints that involve variables that

may be affected by new constraints hence needs to perform a transitive reachability computation in the constraint graph.

For CHR full abstraction requires the execution of the tabled predicate with an empty constraint store. If the call environment constraint store were used, interaction with new constraints would violate the assumption of full abstraction.

Full constraint abstraction can be with the source-to-source transformation explained next. The source-to-source transformation is driven by the following user declaration:

```
:- table_chr p(_,chr) with Options.

p(X,Y) :- ...
```

meaning that the predicate `p/2` should be tabled, its first argument is an ordinary Prolog term and its second argument is a CHR constraint variable of which all the constraints are abstracted away.

The predicate is transformed into two separate predicates, where the first one is called, takes care of the abstraction, calls the second predicate and afterwards combines the answer with the previously abstracted away constraints.

```
p(X,Y) :-
        current_chr_store(CallStore)
        set_empty_chr_store,
        tabled_p(X,Y1,AnswerStore),
        set_chr_store(CallStore),
        insert_answer_store(AnswerStore),
        Y1 = Y.

:- table tabled_p/3.
tabled_p(X,Y,S_A) :- ...
```

The complete implementation of `tabled_p` will be discussed in the next section. The given answer constraints are merged into the current global CHR constraint store by the predicate `insert_answer_store/1`. Given the naive representation discussed in the previous section, this boils down to calling a list of goals to impose the constraints.

8.3.4 Answer Projection

Often one wants to project the answer constraint store on the non-local variables of the call. The usual motivation is that constraints on local variables are meaningless outside of the call. The constraint system should be complete so that no unsatisfiable constraints can be lost through projection.

For tabling there is an additional and perhaps even more pressing motivation for projection: a predicate with an infinite number of different answers may be

turned into one with just a finite number of answers by throwing away the constraints on local and unreachable variables.

Example 8.2 Consider this program:

```
path(From,To,X) :-
        edge(From,To,X).
path(From,To,X) :-
        path(From,Between,X), path(Between,To,X).

edge(a,a,X) :-
        leq(X,Y),
        leq(Y,1).

leq(X,X)              <=> true.
leq(X,Y) \ leq(Y,X)   <=> X = Y.
leq(X,Y) \ leq(X,Y)   <=> true.
leq(X,Y) , leq(Y,Z)   ==> leq(X,Z).
```

It defines a `path/3` predicate that expresses reachability in a graph represented by `edge/3` predicates. The first two arguments of both predicates are edges (origin and destination) and the third is a constraint variable. Along every edge in the graph some additional constraints may be imposed on this variable. In our example, the graph consists of a single loop from edge `a` to itself. This loop imposed two less-than-or-equal-to constraints: `leq(X,Y), leq(Y,1)`. The variable `Y` is a local variable and the fourth rule for `leq/2` derives that `leq(X,1)` also holds.

There are an infinite number of paths in our simple graph, one for each non-zero integer n. A path for n takes the loop n times. For every time the loop is taken a new variable $\texttt{Y}_i$ is created and two more constraints `leq(X,`$\texttt{Y}_i$ and `leq(`$\texttt{Y}_i$`,1)` are added. Through the propagation rule also an `leq(X,1)` is added for each time the loop is taken. The second simpagation rule however removes all but one copy of this last constraint.

Even though there are an infinite number of answers, the constraints involving the local variables $\texttt{Y}_i$ are of no interest and only the single `leq(X,1)` is relevant.

In general constraint projection onto a set of variables transforms a constraint store into another constraint store in which only variables of the given set are involved. The form of the resulting constraint store strongly depends on the particular constraint solver and its computation may involve arbitrary analysis of the original constraint store.

We propose an elegant CHR-based approach to projection. It is a rather compact and high level notation and as such it might be possible to infer conditions on its usage under which the technique is provably correct.

The user declares the use of the CHR-based approach to projection with this declaration:

```
:- table_chr p(_,chr) with [projection].
```

and implements the projection as a number of CHR rules that involve the special `project/1` constraint. The `project/1` constraint has as its argument the set of variables to project on.

The source-to-source transformation generates the predicate `tabled_p` based on the declaration:

```
tabled_p(X,Y,S_A) :-
        orig_p(X,Y)
        project([Y]),
        extract_store_representation(S_A).
```

Here the predicate `extract_store_representation/1` converts from the ordinary global store representation to the naive tabled store representation, discussed in Section 8.3.2. The `project/1` constraint is called after the execution of the original code `orig_p/2` of the predicate `p/2`. Its purpose is to rewrite the constraint store into its projected form.

To implement the projection simpagation rules can be used to look at and decide what constraints to remove. A final simplification rule at the end can be used to remove the `project/1` constraint from the store.

The following example shows how to project away all `leq/2` constraints that involve arguments not contained in a given set `Vars`:

```
project(Vars) \ leq(X,Y) <=>
                \+ (member(X,Vars),member(Y,Vars)) | true.
project(Vars)            <=> true.
```

Besides removal of constraints more sophisticated operations such as weakening are possible. E.g. consider a set solver with two constraints: `in/2` that requires an element to be in a set and `nonempty/1` that requires a set to be non-empty. The rules for projection could include the following weakening rule:

```
project(Vars) \ in(Elem,Set) <=>
                member(Set,Vars),
                \+ member(Elem,Vars) | nonempty(Set).
```

8.3.5 Entailment checking and other answer combinations

Some of the answers computed for a tabled predicate may be redundant and so need not be saved. The property is exploited by the optimized definition of answer sets, Definition 8.5. Consider for example that the answer `p(a,X)` is already in the

table of predicate `p/2`. Now a new answer, `p(a,b)` is found. This new answer is redundant as it is covered by the more general `p(a,X)` that is already in the table. Hence it is logically valid to not record this answer in the table, but to simply discard it. This does not affect the soundness or completeness of the procedure.

The idea of this answer subsumption technique is to reduce the number of answers in the table by replacing two (or more) answers by a single answer. Logically, the single new answer has to be equivalent to the disjunction of the replaced answers. For ordinary tabled Prolog, each answer H_i can be seen as a Herbrand constraint $Answer = Term$, e.g. $Answer = p(a, X)$. Now, for any two of these Herbrand constraints H_0 and H_1 the following two properties hold:

1. If the disjunction is equivalent to another constraint, that constraint is equivalent to one of the two constraints.

$$\mathcal{H} \models \exists H : H \leftrightarrow H_0 \vee H_1 \Longleftrightarrow \exists i \in \{0, 1\} : H \leftrightarrow H_i$$

2. If the conjunction is equivalent to one of the two constraints, the disjunction is equivalent to the other.

$$\mathcal{H} \models \exists i \in \{0, 1\} : H_0 \wedge H_1 \leftrightarrow H_i \Longleftrightarrow H_0 \vee H_1 \leftrightarrow H_{1-i}$$

These two properties suggest a possible strategy to compute the equivalent single answer of two answers: check whether the conjunction of two answers is equivalent to one of the two, then the other is the single equivalent answer. Otherwise, there is no single equivalent answer.

We can extend the logically sound idea of answer subsumption to CHR constraints. This path length computation will serve as an illustration:

```
dist(A,B,D) :- edge(A,B,D1), leq(D1,D).
dist(A,B,D) :- dist(A,C,D1), edge(C,B,D2), leq(D1 + D2, D).
```

Suppose appropriate rules for the `leq/2` constraint in the above program, where `leq` means less-than-or-equal. The semantics are that `dist(A,B,D)` holds if there is a path from `A` to `B` of length less than or equal to `D`. In other words, `D` is an upper bound on the length of a path from `A` to `B`.

If the answer `dist(n1,n2,D) :- leq(d1, D)` is already in the table and a new answer `dist(n1,n2,D) :- leq(d2, D)`, where `d1 =< d2`, is found, then this new answer is redundant. Hence it can be discarded. This does not affect the soundness, since logically the same answers are covered.

Operationally, the same answer subsumption strategy as for tabled Prolog can be used to reduce two answer constraint stores S_0 and S_1 to a single answer store S. At the end of the tabled predicate we merge a previous answer store S_0 with a new answer store S_1. After merging the store will be simplified and propagated to S by the available rules of the CHR program $\mathcal{P}$. This combines the two answers

into a new one. This mechanism can be used to check entailment of one of both answers by the other: if the combined answer store S is equal to one of the two, then that answer store entails the other:

$$[\![\mathcal{P}]\!] \models \exists i \in \{0,1\} : S_0 \wedge S_1 \leftrightarrow S_i \Longrightarrow S_{1-i} \leftrightarrow S_0 \vee S_1$$

A sound approximation of the equivalence check for the first equivalence sign in the above formula is syntactical equality. This is our default approach. Below we specify an option that allows for computing a canonical form first, before checking syntactical equality.

The predicate `insert_answer_store/1`, mentioned in Section 8.3.3, is used for the conjunction of two constraint stores. We assume that one store is the current global CHR constraint store.

When the above two answers of the `dist/3` predicate are merged, the following rule `leq/2` rule will simplify the constraint store to retain the more general answer:

```
leq(X,D1) \ leq(X,D2) <=> D1 =< D2 | true.
```

Note that the `dist/3` program would normally generate an infinite number of answers for a cyclic graph, logically correct but not terminating. However, if it is tabled with answer subsumption, it does terminate for non-negative weights. Not only does it terminate, it only produces one answer, namely `dist(n1,n2,D) :- leq(d,D)` with `d` the length of the shortest path. Indeed, the predicate only returns the optimal answer.

The above strategy is a sound approach to finding a single constraint store that is equivalent to two others. However, it is not complete: a single constraint store may be equivalent to the disjunction of two others, while it is not equivalent to one of the two. This is because the first property for the Herbrand constraints does not hold for all constraint solvers, e.g. $leq(X,Y) \vee leq(Y,X) \leftrightarrow true$. Nevertheless it is a rather convenient strategy, since it does not require any knowledge on the particularities of the used constraint solver. That makes it a good choice for the default strategy for CHR answer subsumption. Better strategies may be supplied for particular constraint solvers.

For some applications one can combine answers with answer generalization which does not preserve the logical correctness. An example in regular Prolog would be to have two answers `p(a,b)` and `p(a,c)` and to replace the two of them with one answer `p(a,X)`. This guarantees (for positive programs) that no answers are lost, but it may introduce extraneous answers. A similar technique is possible with constrained answers. While this approach is logically unsound, it may be acceptable for some applications if the overall correctness of the program is not affected. An example is the use of the least upper bound operator to combine answers in the tabled abstract interpretation setting of (Codish, Demoen, and Sagonas 1998).

Summary Two additional options can be supplied to extend the automatic transformation:

- `canonical_form(`*`PredName`*`)` specifies the name of the predicate that should compute the (near) canonical form of the answer constraint store. This canonical form is used to check equivalence of two constraint stores.
- `answer_combination(`*`PredName`*`)` specifies the name of the predicate that should compute the combination of two answers, if they can be combined. The value `default` selects the above mentioned default strategy.

A stronger, semantically founded technique for implication checking is developed in Chapter 9.

A subsumption-based optimization technique The technique used in the `dist/3` program is to replace the computation of the exact distance of a path with the computation of an upper bound on the distance via constraints. Then, by tabling the predicate and performing answer subsumption, the defining predicate has effectively been turned into an optimizing one, computing the length of the shortest path. It is a straightforward yet powerful optimization technique that can be applied to other defining predicates as well, turning them into optimizing predicates with a minimum of changes. The usual approach consists in computing the list of all answers, using the `findall/3` meta-programming built-in, and in processing this list of answers.

8.3.6 Evaluation of a shipment problem

Problem statement: *There are N packages available for shipping using trucks. Each package has a weight and some constraints on the time to be delivered. Each truck has a maximum load and a destination. Determine whether there is a subset of the packages that can fully load a truck destined for a certain place so that all the packages in this subset are delivered on time.*

The problem is solved by the *truckload* program:

Listing 8.1 - The truckload Program

```
:- constraints leq/2.

leq(X,X)              <=> true.
leq(N1,N2)            <=> number(N1), number(N2) | N1 =< N2.
leq(N1,X) \ leq(N2,X) <=> number(N1), number(N2), N1 > N2 | true.
leq(X,N1) \ leq(X,N2) <=> number(N1), number(N2), N1 < N2 | true.
leq(X,Y)  \ leq(X,Y)  <=> true.
leq(X,Y)  , leq(Y,Z)  ==> leq(X,Z).

truckload(0,0,_,_).
```

```
truckload(I,W,D,T) :-                    % do not include pack I
        I > 0,
        I1 is I - 1,
        truckload(I1,W,D,T).
truckload(I,W,D,T) :-                    % include pack I
        I > 0,
        pack(I,Wi,D,T),
        W1 is W - Wi,
        W1 >= 0,
        I1 is I - 1,
        truckload(I1,W1,D,T).

pack(30,29,chicago,T) :- leq(19,T),leq(T,29).
pack(29,82,chicago,T) :- leq(20,T),leq(T,29).
pack(28,24,chicago,T) :- leq(8,T),leq(T,12).
%...
pack(3,60,chicago,T) :- leq(4,T),leq(T,29).
pack(2,82,chicago,T) :- leq(28,T),leq(T,29).
pack(1,41,chicago,T) :- leq(27,T),leq(T,28).
```

Packages are represented by clauses of `pack/4`, e.g.

```
pack(3,60,chicago,T) :- leq(4,T),leq(T,29).
```

means this is the third package, it weights 60 pounds, is destined for Chicago and has to be delivered between the 4th and the 29th day. The `truckload/4` predicate computes the answer to the problem, e.g. `truckload(30,100,chicago,T)` computes whether a subset of the packages numbered 1 to 30 exists to fill up a truck with a maximum load of 100 pounds destined for Chicago. The time constraints are captured in the bound on the constraint variable `T`. There may be multiple answers to this query, if multiple subsets exist that satisfy it.

We have run the program in four different modes:

- Firstly, the program is run as is without tabling.
- Secondly, to avoid the recomputation of subproblems in recursive calls the `truckload/4` predicate is tabled with:

```
:- table_chr truckload(_,_,_,chr)
             with [representation(naive)].
```

- In a third variant the answer store is canonicalized by simple sorting such that permutations are detected to be identical answers:

load	no tabling	tabling		
		plain	sorted	combinator
100	¡1	100	100	100
200	160	461	461	451
300	2,461	1,039	1,041	971
400	12,400	1,500	1.510	1,351
500	$>$ 5 min.	1,541	1,541	1,451

Table 8.5: Runtime results for the `truckload` program

load	tabling		
	plain	sorted	combinator
100	286	286	279
200	979	956	904
300	1,799	1,723	1,584
400	2,308	2,202	2,054
500	2,449	2,365	2,267

Table 8.6: Space usage for the `truckload` program

```
:- table_chr truckload(_,_,_,chr)
             with [representation(naive),
                   canonical_form(sort)].
```

- Finally, in the fourth variant we apply a custom combinator to the answers: two answers with overlapping time intervals are merged into one answer with the union of the time intervals. For example the disjunction of the following two intervals on the left is equivalent to the interval on the right:

$$(1 \leq T \leq 3) \quad \vee \quad (2 \leq T \leq 4) \quad \longleftrightarrow \quad (1 \leq T \leq 4)$$

  This variant is declared as, with `interval_union/3` the custom answer combinator:

```
:- table_chr truckload(_,_,_,chr)
             with [representation(naive),
                   answer_combination(interval_union)].
```

Table 8.5 contains the runtime results of running the program in the four different modes for different maximum loads. Runtime is in milliseconds and has been obtained on an Intel Pentium 4 2.00 GHz with 512 MB of RAM. For

	tabling		
load	plain	sorted	combinator
100	324	324	283
200	2,082	2,069	1,686
300	4,721	4,665	3,543
400	5,801	5,751	4,449
500	4,972	4,935	4,017

Table 8.7: Number of tabled answers for the `truckload` program

the modes with tabling the space usage, in kilobytes, of the tables and number of unique answers have been recorded as well, in Table 8.6 and Table 8.7 respectively.

It is clear from the results that tabling does have an overhead for small loads, but that it scales much better. Both the modes with the canonical form and the answer combination have a slight space advantage over plain tabling which increases with the total number of answers. There is hardly any runtime effect for the canonical form, whereas the answer combination mode is faster with increasing load.

In summary, tabling can be useful for certain programs with CHR constraints to considerably improve scalability. Canonicalization of the answer store and answer combination can have a favorable impact on both runtime and table space depending on the particular problem.

8.4 Conclusion

In this chapter we have shown how to integrate the committed choice bottom-up execution of CHRs with the tabled top-down execution of XSB. Our implementation realizes SLG$^{\mathcal{D}}$ resolution in terms of SLG. In particular the issues related to the consistency of the global CHR store and tables have been established and solutions have been formulated for call abstraction, tabling constraint stores, answer projection, answer combination (e.g. for optimization), and answer entailment checking.

Part of this work was published at the International Conference of Logic Programming (Schrijvers and Warren 2004) and the Colloquium on Implementation of Constraint and Logic Programming Systems (Schrijvers, Warren, and Demoen 2003). At the former occasion, this work received the *Best Paper Award*. Finally, we would like to mention that an XSB release, number 2.7, with the presented CHR system integrated with tabling is publicly available since December 30, 2004 (see `http://xsb.sf.net`).

8.4.1 Related and Future Work

Ad hoc approaches to using constraints in XSB were used in the past, such as a meta-interpreter (Mukund, Ramakrishnan, Ramakrishnan, and Verma 2000), interfacing with a solver written in C (Du, Ramakrishnan, and Smolka 2000) and explicit constraint store management in Prolog (Pemmasani, Ramakrishnan, and Ramakrishnan 2002). However, these approaches are quite cumbersome and lack the ease of use and generality of CHR.

The theoretical background for this chapter, SLG$^{\mathcal{D}}$ resolution, was realized by Toman in (Toman 1997). Toman establishes soundness, completeness and termination properties for particular classes of constraint domains. While he has implemented a prototype implementation of SLG$^{\mathcal{D}}$ resolution for evaluation, no practical and fully-fledged implementation in a Prolog system was done.

The most closely related implementation work that this chapter builds on is (Cui and Warren 2000a), which presents a framework for constraint solvers written with attributed variables. Attributed variables are a much cruder tool for writing constraint solvers though. Implementation issues such as constraint store representation and scheduling strategies that are hidden by CHR become the user's responsibility when she programs with attributed variables. Also in the tabled setting, the user has to think through all the integration issues of the attributed variables solver. For CHR we have provided generic solutions that work for all CHR constraint solvers and more powerful features can be accessed through parametrized options.

Guo and Gupta propose a technique for dynamic programming with tabling ((Guo and Gupta 2003)) that is somewhat similar to the one proposed here. During entailment checking a particular argument in a new answer is compared with the value in the previous answer. Either one is kept depending on the optimization criterion. Their technique is specified for particular numeric arguments whereas ours is for constraint stores and as such more general. Further investigation of our technique is certainly necessary to establish the extent of its applicability.

In (Schrijvers, Warren, and Demoen 2003) we briefly discuss two applications of CHR with tabling in the field of model checking. The integration of CHR and XSB has shown to make the implementation of model checking applications with constraints a lot easier. The next step in the search for applications is to explore more expressive models to be checked than are currently viable with traditional approaches.

Further applications should also serve to improve the currently limited performance assessment of CHR with tabling. The shipment problem has given us some indication of improved performance behavior in practice, but theoretical reasoning indicates that slow-downs are a possibility as well.

Partial abstraction and subsumption are closely related. The former transforms a call into a more general call while the latter looks for answers to more general calls, but if none are available still executes the actual call. We still have to

look at how to implement partial abstraction and the implications of variant and subsumption based tabling.

Finally, better automatic techniques for entailment testing, such as those of Chapter 9, and for projection should be investigated in the context of SLG$^{\mathcal{D}}$.

Chapter 9

Automatic Implication Checking for CHR Solvers

9.1 Introduction

In this chapter we investigate how to automatically extend a CHR constraint solver not only to answer questions of satisfiability, but also to answer questions about implication. Such questions of implication allow for the extraction of information from the logical meaning of a final CHR state beyond the syntactical.

Example 9.1 Consider the following equality constraint solver that we will use as an example throughout this chapter. The following four CHR rules define an equality solver *eq*, with `eq/2` the equality constraint:

```
reflexive  @ eq(X,Y) <=> X == Y | true.
redundant  @ eq(X1,Y1) \ eq(X2,Y2) <=> X1 == X2, Y1 == Y2 | true.
symmetric  @ eq(X,Y) ==> eq(Y,X).
transitive @ eq(X1,Y1), eq(X2,Y2) ==> Y1 == X2 | eq(X1,Y2).
```

with the arguments of `eq/2` variable identifiers and `==/2` syntactic identity.

Consider the query `eq(a,b)` for which the constraint store in the final state contains the constraints `eq(a,b),eq(b,a)`. Answering whether the constraint `eq(b,a)` is implied may seem trivial: it is present in the constraint store. However, answering the constraint `eq(a,a)` may seem less so: it is not present in the constraint store, but nonetheless it is implied by the logical theory of the program.

Note that, unlike in Prolog, an implication check is not performed by simply calling the constraint. When a predicate is called in Prolog a proof tree is constructed that shows that the predicate is implied by the Prolog program. However, when calling a (CHR) constraint, it is added to the constraint store and further

restricts possible solutions. While failure in Prolog means that a predicate is not implied (or implication is not provable), failure in a constraint solver means that the constraint store is inconsistent and no solution exists.

The above example already illustrates the basic notion of implication checking. For more complicated constraint solvers questions of implication may be more complex and less obvious to answer.

Usually a question of implication of a constraint c is answered with the following *copying approach*. A copy C' is made of the query C and final states for C and $C' \wedge c$ are compared. If the two final states are equivalent, then c is implied by C. This approach is not complete for non-canonical solvers (see Section 4.4.1).

Our novel *trailing approach* does not copy the entire constraint store, but performs the check in place using a trailing mechanism. In order to provide this implication checking functionality, the original CHR program is transformed automatically with a source-to-source transformation. This method is extended to work for hierarchically organized modular CHR solvers, which is important for the maintenance and evolution of multi-solver applications.

We show the soundness of our trailing method and its completeness for a restricted class of canonical solvers as well as for specific existing non-canonical CHR solvers. Also a comparison is made of the copying and trailing approaches.

Overview In the next section, the necessary CHR constraint solver notions are introduced. Section 9.3 presents the basic method of implication checking and soundness as well as completeness results. Section 9.4 extends this technique to hierarchical solvers. The completeness of implication checking with our method is studied for several concrete CHR solvers in Section 9.5. Section 9.6 presents some experimental results. Finally, Section 9.7 concludes.

9.2 CHR Solvers

We will use the symbol *CS* to denote a CHR program $\mathcal{P}$ that is a constraint solver. Typically, for these CHR solvers the logical meaning (see Section 4.2) is important: it models the constraint theory that the program implements.

The logical theory of a CHR solver *CS* assigns a logical meaning $\mathit{meaning}(\sigma)$ to every possible execution state σ. As we are mostly interested in these logical meanings throughout this chapter, we will lift all previously defined notions regarding execution states to the level of these logical meanings for reasons of brevity. If the reader is interested in formulations on the level of execution states, she need only replace every occurrence of a logical meaning C with either σ or $\mathit{meaning}(\sigma)$ depending on the context, where σ is the corresponding execution state.

Given the logical theory of the solver $[\![CS]\!]$ it is possible to formally prove various theorems. The object of this chapter is to prove implication theorems: given a logical meaning C verify whether some simple constraint c is implied, i.e. $[\![CS]\!] \models C \rightarrow c$. Implication checking is a very useful notion: it allows for the interpretation of C with respect to some property of interest c. Many popular constraint solvers provide implication checking in one form or another, e.g. the *conditional* constraint combinator of Mozart (Schulte 2000), the reified constraints of the `clp(FD)` library and the `entailed/1` predicate of the `clp(QR)` library in SICStus (Intelligent Systems Laboratory 2003).

9.2.1 Required Notions

It will be possible to derive completeness results of implication checking for the class of range-restricted CHR-only solvers. The properties of this class are:

- The notion of canonicity (as well as confluence) is defined in Section 4.4.
- CHR-only solvers are solvers with a syntactical restriction: the rule bodies of the solvers do not contain any built-in constraints.
- Range-restrictedness. A CHR solver *CS* is range-restricted, if for every CHR rule in the solver any grounding of the head of the rule is also a grounding of its body. In (Stuckey and Sulzmann 2005) range-restrictedness is a sufficient condition for canonicity.

The following lemma combines the properties of CHR-only and range-restricted solvers and will be used in Section 9.3.

Lemma 9.1 (Range-Restricted CHR-Only Solvers)
If a CHR-only solver CS is range-restricted, then:

$$\forall C : C \rightarrowtail_{CS} C' \Rightarrow vars(C') \subseteq vars(C)$$

Proof: Obvious from the definitions of "range-restricted" and "CHR-only". □

Example 9.2 The equality solver *eq* defined in Example 9.1 is a confluent solver. For example, for the query `eq(a,a)` a single application of the *reflexive* rule or one application of the *symmetric* and two of the *reflexive* rule both yield an empty constraint store.

The equality solver *eq* is even a canonical range-restricted solver. It is obvious that it is range-restricted. Showing that it is canonical relies on showing that it returns a store $\{eq(x,y), eq(y,x) \mid$ x and y are distinct nodes connected in the graph created by all the *eq* constraints in the goal$\}$.

9.3 Basic Implication Checking

In this section we present our basic technique for implication checking in a stand-alone CHR solver. It is extended to CHR solver hierarchies in the next section.

First, in Section 9.3.1, we derive an approach on a theoretical level and establish its soundness and completeness. Next, in Section 9.3.2, we present two concrete approaches that realize the technical approach: a naive copy approach, that serves as a reference for comparison, and our trailing approach.

9.3.1 Theoretical Approach

Based on the properties of logical implication and conjunction, we can use the following technique to verify whether a constraint c is implied by a conjunction of constraints C.

$$[\![CS]\!] \models C \rightarrow c \quad \Leftrightarrow \quad [\![CS]\!] \models (C \wedge c) \leftrightarrow C$$

Namely we can use the equivalence of the conjunctions C and $C \wedge c$ to conclude implication.

Solved Forms

As the solved forms $solve(C)$ and $solve(C \wedge c)$ are equivalent to C and $C \wedge c$ respectively, the equivalence test may be performed on them.

The following soundness result holds:

Theorem 9.1 (Soundness) *For any CHR solver CS:*

$$\forall C, c : [\![CS]\!] \models \bar{\exists}_{vars(C \wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C \wedge c)} solve(C \wedge c) \quad \Rightarrow \quad [\![CS]\!] \models C \rightarrow c$$

Proof: We have that

$$[\![CS]\!] \models C \leftrightarrow \bar{\exists}_{vars(C \wedge c)} solve(C)$$

and

$$[\![CS]\!] \models C \wedge c \leftrightarrow \bar{\exists}_{vars(C \wedge c)} solve(C)$$

from the soundness of CHR, Theorem 4.1. Hence if

$$[\![CS]\!] \models \bar{\exists}_{vars(C \wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C \wedge c)} solve(C \wedge c)$$

we have that $[\![CS]\!] \models C \leftrightarrow (C \wedge c)$ and hence $[\![CS]\!] \models C \rightarrow c$. □

Completeness only holds if the solved forms exist. This is the case for terminating solvers:

Theorem 9.2 (Terminating Completeness) *If the CHR solver CS is terminating, then implication checking is complete. That is*

$$\forall C, c : [\![CS]\!] \models C \rightarrow c \quad \Rightarrow \quad [\![CS]\!] \models \bar{\exists}_{vars(C \wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C \wedge c)} solve(C \wedge c)$$

Proof: Direct from Definition 4.8 in Section 4.4 and the soundness of CHR. □

Syntactical Equivalence

In practical implementations, logical equivalence testing ($\bar{\exists}_{vars(C\wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C\wedge c)} solve(C\wedge c)$) of projected constraint is restricted to syntactic equivalence of multisets ($solve(C) \equiv solve(C \wedge c)$). The latter is straightforward, while the former may require general theorem proving. For range-restricted CHR solvers the two tests are equivalent since $vars(solve(C)) \subseteq vars(C\wedge c)$ and $vars(solve(C\wedge c)) \subseteq vars(C \wedge c)$.

Theorem 9.3 (Specific Soundness) *For a range-restricted CHR-only solver CS:*

$$\forall C, c : [\![CS]\!] \models solve(C) \equiv solve(C \wedge c) \quad \Rightarrow \quad [\![CS]\!] \models C \rightarrow c$$

Proof: The theorem is derived from Theorem 9.1. The two projections in Theorem 9.1 are omitted for range-restricted CHR-only solvers based on Lemma 9.1. The logical equivalence ($\leftrightarrow$) is replaced with syntactical equivalence ($\equiv$) based on the definition of canonical solvers 4.9 in Section 4.4.1. □

The syntactical equivalence approach of implication checking is complete for canonical range-restricted CHR-only solvers.

Theorem 9.4 (Specific Completeness) *If CS is a range-restricted CHR-only solver, then implication checking is complete. That is*

$$\forall C, c : [\![CS]\!] \models C \rightarrow c \quad \Rightarrow \quad [\![CS]\!] \models solve(C) \equiv solve(C \wedge c)$$

Proof: From Definition 4.9 in Section 4.4.1 we have that

$$\forall C, c : [\![CS]\!] \models C \leftrightarrow (C\wedge c) \Rightarrow [\![CS]\!] \models \bar{\exists}_{vars(C\wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C\wedge c)} solve(C\wedge c)$$

Since CS is range-restricted $vars(solve(C)) \subseteq vars(C\wedge c)$ and $vars(solve(C\wedge c)) \subseteq vars(C \wedge c)$. Hence $solve(C) \equiv solve(C \wedge c)$. □

Note that in the special case that $C \wedge c$ fails, c is not implied by C as $C \wedge c$ is not satisfiable. This case is correctly covered by our approach.

A constraint solver does not have to be canonical for our implication checking to be complete. In Section 9.5 we will show that our method is also complete for several non-canonical, even non-confluent, CHR solvers.

9.3.2 Practical Approaches

Copy Approach

The straightforward implementation approach for implication checking is to make a copy C' of C, compute $C'' = solve(C' \wedge c)$ and then check syntactical equivalence of C with C''. We call this approach the *copy approach*.

Trailing Approach

The above copying approach may be quite expensive. C may consist of two parts $C = C_1 \wedge C_2$ such that C_1 is a minimal set of constraints that imply c. By copying C in its entirety C_2 is copied unnecessarily and causes undue overhead in the final equivalence test.

We propose the *trailing* approach for CHR-only solvers. It only looks at a minimal set of constraints: the conjunction $C \wedge c$ is solved in place and a trail of changes is maintained. Analysis of the trail afterwards tells us whether the resulting store is equivalent to the original. If that is the case, the updates to the store may remain. Otherwise, the trail is used to revert to the original situation.

Example 9.3 The following CHR rule conceptually represents the above strategy. Keep in mind the refined semantics with sequential left-to-right execution of the constraints and top to bottom trial of rules.

```
implication @ check_eq(X,Y) <=> eq(X,Y), analyse_trail.
```

The `check_eq/2` constraint represents the implication check. Calling this constraint will either succeed or fail, since `analyse_trail/0` succeeds if the resulting store is equivalent and fails if it is not.

Our trail analysis has to inspect the addition and removal of constraints to decide equivalence. Roughly, if any constraint is added or deleted during the implication checking, the resulting store will not be equivalent to the original. More precisely, stores are also equivalent if a constraint is only *temporarily* added or deleted, since addition and deletion of the same constraint cancel each other out.

The following set of CHR rules reflects this approach for `analyse_trail/0`:

```
temporary @ analyse_trail \ added(C), removed(C) <=> true.
addition  @ analyse_trail \ added(C) <=> fail.
removal   @ analyse_trail \ removed(C) <=> fail.
success   @ analyse_trail <=> true.
```

Here `added/1` and `removed/1` represent trail entries of added and deleted constraints.

The code above only works correctly under the refined operational semantics. A call to `analyse_trail` looks for matching `added/1` and `removed/1` constraints, and removes them using the first rule. If any (unmatched) `added/1` and `removed/1` constraints remain, the second or third rule causes it to fail. Otherwise it reaches the fourth rule which simply succeeds.

In general the original solver is transformed to maintain information about changes with this source-to-source transformation scheme:

Entity	New Rule
p	$p(\bar{x})$ `==>` $added(p(\bar{x}))$ add to front of program
$H_k \setminus H_r$ `<=>` G `|` B	$H_k \setminus H_r$ `<=>` G `|` $removed(p_1(\bar{x}_1)), \ldots, removed(p_n(\bar{x}_n)), B$ replace old rule

with p a constraint predicate and $H_r = [\ p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)]$.

The transformed solver program obtained from the above rules has the disadvantage that it always trails. The following set of rules enable explicit trailing during implication checking only. These rules require `trail_off` to be in the constraint store initially.

```
implication   @ check_eq(X,Y) <=> enable_trail,
                                       eq(X,Y), analyse_trail,
                                   disable_trail.
enable        @ trail_off, enable_trail <=> trail_on.
disable       @ trail_on, disable_trail <=> trail_off.
filter_add    @ trail_off \ added(C) <=> true.
filter_remove @ trail_off \ removed(C) <=> true.
```

Example 9.4 The `eq/2` solver of Example 9.1 is transformed using the general scheme to explicitly generate the necessary `added/1` and `removed/1` constraints:

```
new         @ eq(X,Y) ==> added(eq(X,Y)).
reflexive   @ eq(X,Y) <=> X == Y | removed(eq(X,Y)).
redundant   @ eq(X1,Y1) \ eq(X2,Y2) <=>
                X1 == X2, Y1 == Y2 | removed(eq(X2,Y2)).
symmetric   @ eq(X,Y) ==> eq(Y,X).
transitive  @ eq(X1,Y1), eq(X2,Y2) ==> Y1 == X2 | eq(X1,Y2).
```

Suppose we want to check whether $c \equiv eq(a,c)$ is implied by $C \equiv eq(a,b) \wedge eq(b,c)$, i.e. we want to check whether

$$[\![eq]\!] \models eq(a,b) \wedge eq(b,c) \rightarrow eq(a,c)$$

holds. We call the goal `trail_off, eq(a,b), eq(b,c), check_eq(a,c)`. The first constraint disables the trailing mechanism, and, with the next two constraints, leads to a store $trail\_off, eq(a,b), eq(b,c), eq(a,c), eq(b,a), eq(c,b), eq(c,a)$. The constraint $check\_eq(a,c)$ enables the trailing mechanism and adds $added(eq(a,c))$ using the *new* rule, then the *redundant* rule succeeds adding `removed(eq(a,c))`.

The call to `analyse_trail` removes both of these using the *temporary* rule, then succeeds using the *success* rule. Finally the trailing mechanism is disabled again.

9.4 Implication Checking for Modular Solver Hierarchies

In this section we extend the implication checking technique of the previous section to modular CHR solver hierarchies, with a stress on the modularity.

Typically a host language provides some notion of a module, which often coincides with a program source file. A module has an interface consisting of exported procedures. This interface is the only part that another module is allowed to rely on, i.e. the other module can only call exported procedures. It is a good practice in software engineering to put a logically coherent functionality in a module. Such a module can possibly be reused in different contexts, it can be updated without affecting other modules as long as the interface remains unchanged or it can be replaced by an entirely different module with the same interface. On the level of compilation, a module can typically be compiled independently from other modules.

Modularity and modules are also an important notion for constraint solvers. A constraint solver is a logically coherent unit of code that should be implemented in its own module. There are several ways in which a constraint solver could interact with other constraint solvers: by calling constraints of the other solver or checking implication of constraints of the other solver. By only relying on module interfaces of solvers (solver interfaces) it becomes possible to easily reuse a solver for many different applications, e.g. used by several different other solvers. Moreover, a solver that is used in some application may be replaced with little effort by another solver with the same interface. This easy replacement is an important advantage for constraint logic programming: as the programmer becomes more familiar with the problem to be solved or the nature of the problem shifts the need for a stronger constraint propagation or a more efficient implementation may become apparent.

There is already good support for the interaction between modular CHR solvers and modular built-in (non-CHR) solvers: built-in constraints may appear in bodies of CHR rules and in guards (see (Duck, Stuckey, García de la Banda, and Holzbaur 2003) for the latter). However, the support for the interaction between two CHR solvers is limited: CHR constraints of another solver may only appear in the bodies of rules.

An obvious way to check for the presence of a constraint of another CHR solver as a requirement for a rule application, the obvious way would be to include that constraint in the non-removed part of the head of that rule. However, this

approach breaks modularity: current CHR compilers require that all the rules in whose head a particular CHR constraint appears are contained in the same module.

In this section we address the above problem and even extend it: we allow for a full implication check of another CHR solver's constraints in the same way as for built-in constraints, by writing those constraints in the guards of a CHR solver's rules without compromising modularity. We will restrict ourselves to acyclic CHR constraint solver hierarchies for reasons of implementation and completeness. It is an important part of future work to lift this restriction.

In a constraint solver hierarchy one solver depends on some solvers that in turn depend on other solvers. We say a solver *depends* on another solver if it uses constraints (in guards and bodies) that are defined in the other solver; the former is called the *parent solver* and the latter the *child solver*.

A *modular* CHR solver is a CHR solver that can be compiled using the interface of its child solvers. In particular, no knowledge of their child solvers is required. With respect to modularity, we add that a parent solver should not know about a child solver's dependencies.

Example 9.5 The following less-than-or-equal-to solver *leq* depends on the *eq* solver:

```
leq_new           @ leq(X,Y) <=> check_eq(X,Y) | true.
leq_antisymmetric @ leq(X1,Y1), leq(X2,Y2) <=>
                      check_eq(X1,Y2), check_eq(X2,Y1) | eq(X1,Y1).
leq_redundant     @ leq(X1,Y1) \ leq(X2,Y2) <=>
                      check_eq(X1,X2), check_eq(Y1,Y2) | true.
leq_transitive    @ leq(X1,Y1), leq(X2,Y2) ==>
                      check_eq(Y1,X2) | leq(X1,Y1).
```

This *leq* solver depends on the *eq* solver in two ways. Firstly, it calls `eq/2` constraints in the body of the `leq_antisymmetric` rule. Secondly, it also uses the `check_eq/2` implication check in the guard of all its rules. As both the constraint and the implication check can be exported from the *eq* solver this does not violate modularity.

The operational semantics of the guard of a CHR rule are not entirely captured yet by our automatic implication check. We call an *event* the addition of a constraint of the child solver or one of the solvers it depends on. A CHR rule application in the parent solver may not succeed immediately because a guard is not satisfied, but an event may cause it to be satisfied at a later point.

The semantics of CHR require that CHR constraints of the parent solver are reactivated in case of an event that now satisfies a previously unsatisfied guard. Typically for built-in solvers, relevant events are provided in the solver interface by the solver programmer together with a mechanism to notify interested parties. See

Section 5.2.3 for a discussion of the implication of this event notification mechanism in Prolog.

The following rules describe the necessary operations for such a mechanism of events and notifications for the *eq* solver:

```
new_event @ eq(X,Y) ==> touched(X), touched(Y).
trigger   @ touched(X), delayed(X,Goal,ID) ==> call(Goal).
end_event @ touched(X) <=> true.
kill_goal @ kill(ID) \ delayed(X,Goal,ID) <=> true.
kill_end  @ kill(ID) <=> true.
```

The *eq* solver provides a `touched(X)` event in its interface, without knowing anything about particular uses. The `new_event` rule generates the `touched` event for every variable involved in a new `eq/2` constraint. Users of the interface, such as the *leq* solver will be notified of these events by calling the `delayed/3` constraint. This constraint supplies a callback goal, that is called when the appropriate `touched/1` event fires and allows the notified party to take due action. The `kill/1` constraint allows for the removal of one or more delayed callbacks, based on an identifier and allows the notified party to no longer receive any events.

The following pseudo-code shows how the *leq* solver subscribes itself to `touched` events. It is pseudo-code because it accesses some internals of the CHR implementation.

```
listen @ leq(X,Y) # CID ==> new_delay_id(ID),
                            delayed(X,reactivate(CID),ID),
                            delayed(Y,reactivate(CID),ID),
                            listening(CID,ID).
```

Here `CID` is the internal identifier of the CHR constraint, part of the suspension representation discussed in Section 5.2.2. This pseudo-rule is executed when the `leq(X,Y)` constraint is first activated. The call to `new_delay_id/1` generates a new notification identifier. With the two calls to `delayed` the *leq* solver will be notified of the relevant events. Upon notification the internal goal `reactivate(CID)` is called which reactivates the corresponding constraint. The call to `listening` internally associates the notification identifier with the corresponding `leq/2` constraint. When the `leq/2` constraint with identifier `CID` is removed, internally the `kill/1` constraint is called on all associated notification identifiers. This avoids reactivation of removed constraints.

Several modifications to the implication checking are now necessary to the original scheme, to accommodate both the hierarchy and the modularity. Below we explain how to do trailing for multiple CHR solvers, how to distinguish between trails of recursively called implication checks and how implication checking should interact with the event mechanism.

9.4.1 Trailing Interface

Because of the hierarchy, during an implication check on a parent solver, constraints in the child solver may be added and deleted. Hence, the parent solver trail mechanism should recursively rely on the child solver trail mechanism. The child solver needs to export the necessary trail operations for this.

Example 9.6 The following set of rules encode the trailing dependency of the `leq/2` solver on the `eq/2` solver:

```
rec_analysis  @ leq_analyse_trail ==> eq_analyse_trail.
rec_enable    @ leq_enable_trail  ==> eq_enable_trail.
rec_disable   @ leq_disable_trail ==> eq_disable_trail.
```

9.4.2 Implication Strata

Because of the hierarchy, an implication checking may recursively perform other implication checks. For example, an implication check of a `leq/2` constraint may require the implication check of a `eq/2` constraint. Our trailing approach does not cover this any more. Indeed, it does not distinguish between those `eq/2` constraints added and deleted during the recursive `eq/2` implication check and those during the top level `leq/2` implication check. A more involved trailing mechanism is needed.

Our solution is to associate with each implication stratum (i.e. level of implication check nesting) a stratum identifier. The top level which is not inside any implication check has stratum identifier 0, an implication check called from top level has identifier -1, etc.

Every constraint is labeled with the stratum it is called in. For example, `eq(X,Y)` becomes `eq(X,Y,S)` if it is called in stratum `S`.

Constraints called in the top-level query are assigned stratum 0. Constraints called in the body of a rule inherit the lowest stratum of any constraints in the head and the implication checking lowers the stratum by one.

Example 9.7 For example, the `leq/2` solver is transformed as follows:

```
leq_new           @ leq(X,Y,S) <=> check_eq(X,Y,S-1) | true.
leq_antisymmetric @ leq(X1,Y1,S1), leq(X2,Y2,S2) <=>
     check_eq(X1,Y2,min(S1,S2)-1), check_eq(X2,Y1,min(S1,S2)-1)
                       | eq(X1,Y1,min(S1,S2)).
leq_redundant     @ leq(X1,Y1,S1) \ leq(X2,Y2,S2) <=>
     check_eq(X1,X2,min(S1,S2)-1), check_eq(Y1,Y2,min(S1,S2)-1)
```

```
                        | true.
leq_transitive    @ leq(X1,Y1,S1), leq(X2,Y2,S2) ==>
     check_eq(Y1,X2,min(S1,S2)-1) | leq(X1,Y1,min(S1,S2)).
```

Now it is possible for the implication trailing operations to work on a single stratum by looking at the stratum identifiers: all the related constraints are extended with their stratum's identifier.

However, the implication checking is no longer complete, if the trailing operations are confined to a stratum. The reason is the `temporary` rule:

```
temporary @ analyse_trail(S) \ added(C,S), removed(C,S) <=> true.
```

This rule only cancels out additions and deletions in the same stratum. What is no longer canceled out, is a constraint added in a higher stratum that is removed in a lower stratum and re-added in that lower stratum.

It is possible to re-establish completeness as follows. With every deletion both the stratum of the deleted constraint and the lowest stratum of any of the head constraints is recorded. The latter is the *cause* of the removal. For example, the `leq_antisymmetric` rule then looks like:

```
leq_antisymmetric @ leq(X1,Y1,S1), leq(X2,Y2,S2) <=>
                        check_eq(X1,Y2,min(S1,S2)-1),
                        check_eq(X2,Y1,min(S1,S2)-1)
                        | removed(leq(X1,Y1),S1,min(S1,S2)),
                          removed(leq(X2,Y2),S2,min(S1,S2)),
                          eq(X1,Y1,min(S1,S2)).
```

These rules deal with this new `removed/3` constraint:

```
temporary @ analyse_trail(S) \ added(leq(X,Y),S),
                                removed(leq(X,Y),S,S)
                               <=> true.
promotion @ analyse_trail(S) \ added(leq(X,Y),S), leq(X,Y,S),
                                removed(leq(X,Y),Sr,S)
                               <=> S < Sr | leq(X,Y,Sr).
```

The `temporary` rule still cancels out addition and deletion within the same stratum, but the `promotion` rule promotes a new constraint to the stratum of the previously deleted constraint. In this way the full power of the basic implication checking is retained for solver hierarchies.

The definition of `check_eq/3` is adjusted accordingly. We can get rid of explicit trail enabling and disabling now that we have the implication strata: stratum 0 never requires trailing and the other strata always do.

```
toplevel_add @ added(_,0)      <=> true.
toplevel_rem @ removed(_,_,0)  <=> true.

implication  @ check_eq(X,Y,S) <=> eq(X,Y,S), eq_analyse_trail(S).
```

9.4.3 Inter-stratum Events

During an implication check that takes place in the child solver an event may be fired waking some parent solver constraints that cause some parent solver constraints to be added or deleted in a higher stratum.

However, it is not necessary for these events to travel across strata. An implication check can safely be resolved without propagating any information to the parent solver in the higher stratum: as a child solver does not depend on the parent solver the outcome of an implication check on the child solver should not require interaction with the parent solver.

The other way around, a higher stratum will never generate any event in the presence of a lower stratum, since it is temporarily suspended while execution goes on in the lower stratum and only disappears after the implication check in the lower stratum has finished and thus the lower stratum is gone altogether.

Hence, it is safe and cheaper for events to only trigger callbacks within the same stratum. The modified event code reflects this:

```
new_event @ eq(X,Y,S) ==> touched(X,S), touched(Y,S).
trigger   @ touched(X,S), delayed(X,Goal,S,ID) ==> call(Goal).
end_event @ touched(X,S) <=> true.

listen    @ leq(X,Y,S) # CID ==> new_delay_id(ID),
                                 delayed(X,reactivate(CID),S,ID),
                                 delayed(Y,reactivate(CID),S,ID),
                                 listening(CID,ID).
```

9.5 Case Studies: Non-Canonical Solvers

We have shown in Section 9.3 that our CHR implication checking is complete for canonical solvers. In this section we investigate the completeness for some classical, non-canonical solvers.

It will turn out that the implication checking is still complete in many cases, or can be made complete with a little customization in particular cases.

9.5.1 Naive Union-Find Equality Solver

In Section 3.2.3 a CHR implementation of the naive union-find algorithm is presented. The `union/2` constraint in that implementation may serve as an equality

constraint.

The naive union-find represents equal variables as nodes in the same tree. Any tree with the same variables in it represents the equality of its elements. There is not one preferred, canonical form. For this reason it is even non-confluent: the order of the `union/2` constraints, decides the shape of the tree.

If two variables are unioned that are already equal, their common tree is not modified, nor are any other constraints deleted or added. However, if two variables are not yet equal, a union will merge their trees into one.

Hence, our implication check is complete for this union-find equality solver.

9.5.2 Optimal Union-Find Equality Solver

Next to the naive algorithm also a CHR implementation of an optimal union-find algorithm is given in Section 3.2.4. This algorithm combines path compression with union-by-rank.

Again, when two variables are not equal, their respective trees are merged (by-rank) and this is detected by our implication method. Also, in case the variables are already equal, path compression may still modify the tree by shortening paths from nodes to the root. Because the compressed tree is not syntactically identical to the initial tree, our implication method will reject it.

Nevertheless it is possible to customize the `trail_analysis/0` rules to overcome this problem and safely allow path compression, while rejecting truly new equalities. Namely, instead of these general rules:

```
addition  @ analyse_trail(S) \ added(C,S) <=> fail.
removal   @ analyse_trail(S) \ removed(C,_,S) <=> fail.
```

only the detection of the removal of a `root` constraint is required to detect the linking of two trees:

```
removal   @ analyse_trail(S) \ removed(root(_,_),_,S) <=> fail.
cleanup1  @ analyse_trail(S) \ added(_,S) <=> true.
cleanup2  @ analyse_trail(S) \ removed(_,_,S) <=> true.
cleanup3  @ analyse_trail(S) \ '~>'(X,Y,S) <=> '~>'(X,Y,S+1).
```

These rules do not consider path compression as a non-equivalence of trees. Indeed, they even will not undo the path compression after the implication check, but promote newly created edges to the higher stratum. Hence the compacter tree representation is retained after a succeeding implication check, making future operations cheaper.

9.5.3 Finite Domain Solver

The following CHR solver is a typical bounds propagation based finite domain solver (see (Frühwirth and Abdennadher 2003)). It maintains bounds consistency

for variables. With every variable `X` a `domain(X,L,U)` constraint is associated that maintains the lower and upper bounds, `L` and `U` respectively, of `X`'s domain. Rule `consistency` ensures that the domain is non-empty and rule `intersect` retains the intersection, if two domains exist for `X`.

Constraint propagators, like `dom_eq/2` and `dom_leq/2`, propagate new domains for the involved variables, each time the domain of any of these variables changes. Propagators for other finite domain constraints can be defined analogously.

```
consistent @ domain(X,L,U) ==> L =< U.
redundant  @ domain(X,L1,U1) \ domain(X,L2,U2) <=>
                    L1 >= L2, U1 =< U2 | true.
intersect  @ domain(X,L1,U1), domain(X,L2,U2) <=>
                    domain(X,max(L1,L2),min(U1,U2)).
eq         @ dom_eq(X,Y), domain(X,LX,UX), domain(Y,LY,UY) ==>
                    domain(X,LY,UY), domain(Y,LX,UX).
leq        @ dom_leq(X,Y), domain(X,LX,UX), domain(Y,LY,UY) ==>
                    domain(X,LX,UY), domain(Y,LX,UY).
```

With these rules and our implication checking it is possible to ask whether `domain(X,L,U)` is implied. Either the checked domain is empty and the check fails correctly or the `redundant` rule holds and the check succeeds, or `intersect` rule will combine the check with the domain already present in the store creating a new, smaller domain. Then the trail analysis discovers the change and fails the check. Otherwise the check correctly succeeds. Hence the implication check is optimal for `domain/3` checks.

As the `dom_eq/2` and `dom_leq/3` propagators are never removed, checking for their implication always fails.

9.6 Experimental Evaluation

In this section the trailing approach is compared with the naive copy approach. For this purpose we consider a particular benchmark for the *eq* solver. $n-1$ constraints `eq(`V_i`,`V_{i+1}`)` are imposed for $1 \leq i < n$. This conjunction of equality constraints we call C. The constraint we test for, c, is `eq(`V_1`,`V_n`)` in one case and `eq(`V_1`,`V_{n+1}`)` in the other. The former test succeeds and the latter fails.

Table 9.1 lists the experimental results in seconds obtained for this benchmark with $n = 20$ using the K.U.Leuven CHR system in SWI-Prolog 5.5.8 on an Intel Pentium 4 2.0GHz with 512MB of RAM. Four different approaches, the copy and trailing approach and an optimized version of each, are compared. The copy+ approach is a small improvement on the copy approach: instead of entirely re-computing $solve(C \wedge c)$ it copies the constraint store $solve(C)$ and simply adds c to it. The trailing+ approach is an improvement of the generated code of the

	Approach	$solve(C)$	$solve(C \wedge c)$	$C \rightarrow c$	
$c =$ `eq(V1,Vn)`	copy	3.86	3.88	7.74	100.0%
	copy+	-	-	3.87	50.0%
	trailing	4.19	4.19	4.19	54.1%
	trailing+	3.87	3.87	3.87	50.0%
$c =$ `eq(V1,Vn+1)`	copy	3.86	4.84	8.70	100.0%
	copy+	-	-	5.05	58.0%
	trailing	4.19	5.35	5.35	61.5%
	trailing+	3.87	4.95	4.95	56.9%
$c =$ `union(V1,Vn)`	copy	1.32	1.33	2.65	100.0%
	copy+	-	-	1.38	52.1%
	trailing	1.62	1.62	1.62	61.1%
	trailing+	1.32	1.32	1.32	49.8%
$c =$ `union(V1,Vn+1)`	copy	1.32	1.33	2.65	100.0%
	copy+	-	-	1.38	52.1%
	trailing	1.62	1.64	1.64	61.9%
	trailing+	1.32	1.32	1.32	49.8%

Table 9.1: Experimental Results

trailing approach: a global boolean variable is used to represent whether trailing is enabled or disabled.

The total time to perform implication checking for the copy approach is equal to the sum of the times for $solve(C)$ and $solve(C \wedge c)$[1]. The copy+ approach is 40–50% faster: the cost of copying a constraint store is negligible compared to recomputing it.

With our trailing approach, the time to compute $C \rightarrow c$ corresponds with the time for $solve(C \wedge c)$. While there is about 10% overhead for ordinary use ($solve(C)$), the trailing approach is clearly superior to the copy approach for implication testing: the trailing approach is only slightly worse than the copy+ approach. The trailing+ approach reduces the trailing overhead almost entirely and it behaves as good as or even better than the copy+ approach.

The succeeding test performs a little better than the failing one because the latter propagates many constraints through the transitivity rule.

The table also lists the results for a similar benchmark using the naive union-find program, now with $n = 5000$. The trailing versions have been specialized to not trail additions and removals of constraints that are never stored. The results are similar as for the *eq* solver.

[1]The time to compare the constraint stores is negligible for this benchmark.

9.6.1 Time and Space Formulas

Time Formulas

The following formulas, based on the implementation of the two improved approaches, approximate well the time to check implication. Let $T_{solve}(C)$ denote the time needed to compute the solved form of C and let $|C|$, called the size of C, denote the number of CHR constraints in C.

$$\begin{aligned} T_{copy+}(C \rightarrow c) &= T_{solve}(C) \\ &\quad + k_1 * |solve(C)| \\ &\quad + T_{solve}(solve(C) \wedge c) \\ &\quad + f(|solve(C)|, |solve(solve(C) \wedge c)|)) \end{aligned}$$

$$\begin{aligned} T_{trailing+}(C \rightarrow c) &= (1 + k_4) * T_{solve}(C) \\ &\quad + (1 + k_5) * T_{solve}(solve(C) \wedge c) \\ &\quad + k_6 \end{aligned}$$

For the copy+ approach, the time to check implication is the sum of the time to compute the solved form of C, the time to copy the solved form, the time to compute the solved form when c is added and the time to compare the two solved forms.

Copying a solved form is done in time proportional, k_1, to the size of the internal representation of the solved form. The size of an internal constraint store representation in the K.U.Leuven CHR system is linear in the number of constraints. In our implementation, comparing two solved forms consists of first computing canonical forms and then comparing these. A canonical form is computed by sorting. Comparing is done in time linear in the number of constraints in the smallest solved form. Hence, $f(n, m) = \mathcal{O}(k_2 * (n * \log(n) + m * \log(m)) + k_3 * \min(n, m))$ where k_2 and k_3 depend on the size of the constraint representations.

For the trailing+ approach, the time to check implication is the sum of the time to compute the solved form of C, the time to compute the solved form when c is added and the time to check whether the trail is empty.

While computing the solved forms, there is a little overhead due to the program transformation. This overhead is taken into account with the factors k_4, when trailing is disabled, and k_5, when trailing is enabled. The cost of checking whether the trail is empty is a constant, k_6.

In our implementation of the trailing+ approach the factors k_4, k_5 and k_6 appear to be very small. In particular, k_4 can be made almost zero. For the copy+ approach the factors k_1, k_2 and k_3 are small too. However, if solved forms are sufficiently large and the time to compute them is linear in their size, then the $\mathcal{O}(n * \log(n))$ time of computing canonical forms may dominate the overall time.

Space Formulas

Similarly, we derive formulas for the minimal space usage, assuming perfect reuse. Let d be the derivation length from $solve(C)$ to $solve(C \wedge c)$ and let $||solve(C)||$ defined as:

$$||solve(C)|| = \max_{i=0}^{n} |C_i|$$

where C is any constraint store and $C = C_0 \rightarrowtail C_1 \ldots \rightarrowtail C_n = solve(C_0)$.

$$S_{copy+}(C \rightarrow c) \quad = \quad \max \left\{ \begin{array}{l} l_1 * ||solve(C)|| \\ (l_1 + l_2) * |solve(C)| \\ l_2 * |solve(C)| + \\ \qquad \max \left\{ \begin{array}{l} l_1 * ||solve(solve(C) \wedge c)|| \\ (l_1 + l_2) * |solve(C \wedge c)| \end{array} \right\} \end{array} \right\}$$

$$S_{trailing+}(C \rightarrow c) \quad = \quad l_1 * ||solve(C \wedge c)|| + l_3 * \mathcal{O}(d)$$

The space used by the copy+ approach is the maximum of the space used in three different computation steps:

1. The first is the maximum space used in the computation of $solve(C)$. This space is proportional, factor l_1, to the maximum number of constraints in any of the intermediate states.
2. The second is the space needed for computing the canonical form of $solve(C)$. The needed space is the sum of the space used by the solved form, i.e. $l_1 * |solve(C)|$, and the spaced used by its canonical form, i.e. $l_2 * |solve(C)|$. The factor l_2 may be of the same magnitude as l_1 if the propagation history is small or it may be much smaller for a large propagation history. In the worst case, the ratio l_1/l_2 is proportional to the derivation length.
3. The third is the spaced used by the canonical form of $solve(C)$, i.e. $l_2 * |solve(C)|$, plus the space used by the further computation with a copy of $solve(C)$. The latter is the maximum of the space needed in two more steps:
   (a) The first is the maximum space used in the computation of $solve(solve(C) \wedge c)$.
   (b) The second is the space needed for computing the canonical form of $solve(C \wedge c)$.

Example 9.8 For example, in the two equality solvers used above, $||solve(C)|| = solve(C)$ for any C and $|solve(C \wedge c)| \geq |solve(C)|$. The space formula can then be simplified to:

$$S_{copy+}(C \rightarrow c) \quad = \quad l_2 * |solve(C)| + (l_1 + l_2) * |solve(C \wedge c)|$$

and $l_1/l_2 = \mathcal{O}(n)$ for the `eq/2` solver and $l_1/l_2 = \mathcal{O}(1)$ for the union-find solver.

The space used by the trailing+ approach is much more easily determined. It is the maximum space used during the computation of $solve(C \wedge c)$ plus the space used by the trail. In the worst case, the space used by the trail is proportional to the derivation length d.

Example 9.9 For the two equality solvers the space formula is:

$$S_{trailing+}(C \rightarrow c) \quad = \quad l_1 * |solve(C \wedge c)| + l_3 * \mathcal{O}(d)$$

In case of the succeeding implication check, the trail contains at most two elements:

$$S_{trailing+}(C \rightarrow c) \quad = \quad l_1 * |solve(C \wedge c)| + l_3 * 2$$

In case of the failing implication check, the trail is indeed proportional to the derivation length d for the `eq/2` solver:

$$S_{trailing+}(C \rightarrow c) \quad = \quad l_1 * |solve(C \wedge c)| + l_3 * d$$

For the union-find solver, even under the failing query the trail contains at most four elements.

The two dominant aspects in the approaches are the sizes of the solved forms and the size of the trail. If the solved forms are large, the copying approach behaves badly. If the trail is large, the trailing approach is not recommended.

9.7 Conclusion

In this chapter we have presented a new approach for automatic implication checking in CHR solvers, which is based on a source-to-source transformation. We have established the soundness of our trailing approach as well as its completeness for the class of range-restricted CHR-only solvers. In addition we have studied the completeness for several existing CHR solvers.

Experimental evaluation shows that a naive implementation of the trailing approach is more efficient than a naive implementation of the copying approach. More involved implementations of both approaches are more or less equally efficient. The trailing approach has the advantage that it is easily portable across CHR platforms and no knowledge of canonical forms of constraint stores is needed.

Our trailing approach has also been extended to CHR solver hierarchies. This extension allows for CHR constraints of one solver to be used in the guards of rules of another solver.

This work is available as a technical report (Schrijvers, Demoen, Duck, Stuckey, and Frühwirth 2005a) and has been presented at the 6th International Workshop on Rule-Based Programming (Schrijvers, Demoen, Duck, Stuckey, and Frühwirth 2005b).

9.7.1 Related Work

Previously, it was already shown how to extend built-in solvers with CHR solvers in (Duck, Stuckey, García de la Banda, and Holzbaur 2003). In this chapter we have added a means to extend CHR solvers with other CHR solvers.

In (Duck, García de la Banda, and Stuckey 2004) a general technique is presented for extending basic ask constraints to implication checks of arbitrary logic formulas. The CHR implication checks presented in this chapter can be extended to arbitrary formulas in that way.

The first related technique for CHR was already sketched in (Frühwirth 1993). However, that technique does not represent an ask constraint, but rather a reified constraint. It also performs its operations in place, but spuriously added constraints are never removed explicitly. Instead, they are kept around until they are removed by the CHR solver. The technique is not safe in the general case: it either allows the removal of constraints present in the initial constraint store or can lead to non-termination if removal is suppressed.

9.7.2 Future Work

In future work we intend to extend implication checking techniques for larger classes of CHR solvers. In particular, we would like to be able to project away local variables in built-in and CHR constraints. An important challenge there is automatic inference of constraint projection for CHR constraint solvers, as opposed to simply have the user supply appropriate projection operations.

In addition, we would like to explore the possibility of automatically inferring specific conditions or events that cause constraints to be implied. This would replace user-supplied events. The combination of automatic implication checks with automatic events allows for automatic reified constraints.

The efficiency of our trailing approach may still be improved by generating more specialized code. For example, it may be derived from analysis that some constraints are never removed. If such a constraint is added during an implication check, the check may fail right away. This could be realized through a rule:

```
added(C) <=> fail.
```

where `C` is replaced with the general form of the never removed constraint. For example, in case of the naive union-find program `node/2` constraints are never removed. Depending on the program an arbitrary amount of work may be avoided in this way.

Chapter 10

Conclusions

10.1 Conclusion

The goal of this thesis was to study various aspect of the CHR language related to program analysis, optimized compilation and extensions of the language's expressive power.

We have conducted our study through the implementation of the K.U.Leuven CHR system, a new CHR system for Prolog. Several novel optimizations as well as a general framework for program analysis in terms of abstract interpretation were formulated and validated in our system. The expressive power of the CHR language was improved by integrating the language with tabled resolution and automatically extending CHR constraint solvers with implication checking capabilities.

In addition, we have illustrated the use of CHR in three different application areas. We have established with our programming pearl that it is possible to implement the union-find algorithm in CHR in a concise and elegant manner and with the best known time complexity.

In Section 10.2 we summarize our specific contributions on different aspects of the CHR language. Section 10.3 lists various possibilities for future work that build upon the work presented in this thesis.

10.2 Contributions

We discuss our specific contributions:

CHR Show Cases In Chapter 3 we presented three distinct applications of CHR: two variants of the union-find algorithm, the JmmSolve framework for Java memory models and an algorithm to compute the well-founded semantics.

These applications illustrate the expressive power of CHR. Our programs are compact, elegant and fairly easy to understand. Our study of the time complexity of the optimal union-find program in Chapter 4 and our implementation of constraint stores in Chapter 6 also showed that the expressive power of CHR need not be a compromise for efficiency: similar time complexity as in imperative languages can be obtained.

Assessment of Theoretical Properties A case study of the union-find programs in Chapter 4 has revealed several issues in the current definition of theoretical properties of CHR programs. The current definition and analysis of confluence is not suitable for practical programs as it does take into account neither the additional assumptions and restrictions imposed by the programmer nor the refined operational semantics. The published time complexity formulas are also of limited use for practical programs as they represent a very crude upper bound for time complexity. We have shown how a much more accurate time complexity bound can be derived for the union-find programs.

Implementation Schema Overview In Chapter 5 we have presented a basic compilation schema for CHR in terms of the refined operational semantics. In addition we have explained the optimizations in the reference implementation by Christian Holzbaur in terms of our schema. We were the first to establish the correctness of two of these optimizations with respect to the refined operational semantics.

The K.U.Leuven CHR system One of the most important realizations is the K.U.Leuven CHR system, a new CHR system for Prolog. It has enabled us to extend the current state-of-the-art of optimized CHR compilation: we have developed new constraint stores based on hash tables, groundness declarations and code specialization for ground constraints and anti-monotonic delay avoidance.

The K.U.Leuven CHR system has considerably increased the number of freely available CHR systems. It is currently included in hProlog, XSB and SWI-Prolog and more systems will be supported in the future. A small number of users have already reported their use of the system for commercial, academic, educational and personal projects.

An Abstract Interpretation Framework for CHR In Chapter 7 we proposed an abstract interpretation framework for CHR. This frameworks allows the formulation of program analyses in a more formal and systematic way than was previously the case. We illustrated the framework by applying it to two different analyses: the CHR-specific late storage analysis and the Prolog-based groundness analysis that was lifted to the level of CHR.

The framework facilitates the construction of more complicated analysis domains and the composition of analyses into more powerful synergistic ones. It also facilitates establishing the correctness of the formulated analyses, an aspect that was previously mostly ignored.

Integration of CHR with Tabling We have realized the integration of CHR with tabled resolution in Chapter 8. In earlier work the semantics of constraint logic programming with tabling had already been proposed and several prototype implementations had already been realized. Our work realized the first high-level and user-friendly integration, which is in the spirit of both tabled resolution and CHR.

We showed the basic compilation scheme, based on a source-to-source transformation, together with various customizations, call abstraction, answer projection, entailment checking and generalized answer combination. This compilation can easily be hidden from the user with a simple declaration-based interface.

Automatic Implication Checking for CHR Solvers In Chapter 9 CHR constraint solvers were extended automatically from answering questions about satisfiability to answering questions about implication. The general soundness and the completeness for a particular class of constraint solvers and several individual solvers were established.

Implication checking is needed for extracting information of interest from a constraint store. It is also an essential building block for composing constraint solvers and realizing complex user-defined constraints in terms of more primitive constraints. We have shown how it can be used to construct modular CHR constraint solver hierarchies, which is an important first step in providing encapsulation and reuse capabilities for the CHR language.

10.3 Future Work

In our work we have touched upon various aspects of the CHR language and made our contributions. At the same time, it was apparent that many challenges in the development of the language still lie ahead.

This section briefly discusses what we believe to be relevant open issues on the level of usability (Section 10.3.1) and efficiency (Section 10.3.2). Improvements on these accounts will make CHR a more suitable language for real-world applications in a wide range of application domains.

10.3.1 Usability Issues

Verification In Chapter 4 we have touched upon several properties that are useful indicators of the correctness of a CHR program with respect to its inten-

ded behavior and meaning. Several analyses exist for verifying these properties. However, our case studies have shown that these analyses often do not yield an accurate result, but rather a gross overestimation on the safe side. This leaves the more fine-grained analysis up to the programmer.

The inaccuracy of the analyses is for a large part due to the fact that they do not take into account the particular intended use of a CHR program. A typical example is that only certain queries are intended to be used. To improve the accuracy of such analyses and make them more useful to programmers, it should be possible for a programmer to specify the intended use together with the program. This specification should then be taken into account by the various analyses in order to improve the accuracy of their results.

The proposed specification of intended use may not only improve the usability of verification analyses, but may also serve as program documentation, i.e. a contract for the users of the program. Type and mode declarations are important parts of such a specification.

Patterns of Reuse Throughout this text we have shown various CHR programs, in particular in Chapter 3. Some of these programs exhibit a similar structure or contain similar programming idioms. However, hardly any work has been done on reusing general programming idioms and code fragments. Currently, reuse of CHR programs or program fragments is mostly limited to copying from one program to another. It is well-known from software engineering that such a code copying approach is the cause of bugs and that it encumbers program maintenance.

The first step towards reuse are the source-to-source transformations proposed in (Frühwirth and Holzbaur 2003). The proposed transformations take an existing CHR program and transform it into a new one with a different operational behavior: probabilistic CHR, fair CHR, ...

However, reuse should be possible for other purposes than just modifying behavior. Useful ideas for reuse tools may come from object-oriented programming: (implementation) inheritance and mixin classes allow for the reuse of method code within different contexts. For CHR a similar idea would be useful for the reuse of generic rules, such as the typical propagation rule that implements transitivity or a set of rules that implements an aggregate function over the constraint store. Such code reuse may conceptually take place at the level of CHR source code while the CHR system may internally be able to generate aggressively optimized code for frequently used patterns it knows about. For example aggregate functions are expressible in terms of CHR rules, but straightforward implementation on top of the CHR constraint store should yield much better performance than naively generated code for the CHR rules.

Another frequent pattern is the use of constraints as boolean flags to switch on or off particular CHR rules.

Constraint Solvers Constraint solvers were the primary intended applications of CHR. Hence, the language has several features that are essential for easily implementing constraint solvers, e.g. its concurrency and re-checking of rules. However, CHR does not provide much more than these primitive facilities, while many constraint solvers share more advanced functionality as well.

In Chapter 9 we have already presented how CHR constraint solvers can automatically be extended with implication checking functionality. Another typical constraint solver functionality is projection. We have already started work on automatic constraint projection with Peter Stuckey and Gregory Duck, though much more work needs to be done to reach a useful result.

While we have provided the first basic support for combining CHR solvers in Chapter 9, much more work remains. For example, on the level of communication between constraint solvers the automatic inference of mutually useful events would take away the burden from the programmer. Another aspect of constraint solver interoperability as well as within a single CHR constraint solver is the matter of priorities. Constraint Logic Programming languages such as ECL^{i}PS^e (IC-Parc) and SICStus (Intelligent Systems Laboratory 2003) allow for different priorities to be associated with different constraint propagators. Propagators with a higher priority are woken first. A similar feature makes sense for CHR too: priorities could be assigned to CHR constraints within one solver or to CHR solvers. Constraints with the highest priority or constraints of the solver with the highest priority would be put first on the execution stack in the **Solve** transition. This takes away some of the current non-determinism in this **Solve** transition and gives the programmer a way to control it.

Host Languages Currently only a limited number of host languages support CHR. Most notably is Prolog, with CHR support in SICStus Prolog, ECL^{i}PS^e, Yap, SWI-Prolog and XSB. Some preparatory work is also in progress to port the K.U.Leuven CHR system to Ciao Prolog. With this coverage of Prolog systems, CHR is available to almost every Prolog programmer.

However, Prolog is not a very popular language in terms of the number of programmers and industrial applications. Currently imperative and object-oriented programming language are much more prevalent. Although a few CHR systems already exist for the fairly popular object-oriented language Java, these systems are currently not state-of-the-art with respect to CHR compiler technology and they are mainly focused on the constraint logic instead of smooth integration into the host language.

It also makes sense to provide systems for popular scripting languages such as Python and Ruby with which CHR shares its rapid prototyping capabilities.

To this end we have already started a joint project with Christian Holzbaur to develop a CHR compiler front-end that compiles CHR programs to intermediate code. It should be possible to compile this intermediate code to any desired host

language. When this front-end is shared among CHR systems for many different languages, any optimizations in the front-end become immediately available for all those systems.

10.3.2 Efficiency Issues

Better Scalability Improved scalability is a critical issue for CHR's suitability in real-world applications. It should be able to cope with vast amounts of data, instead of only moderately sized problems.

Our work and that of (Holzbaur, García de la Banda, Stuckey, and Duck 2005) have initiated scalability improvements through optimized compilation of CHR with various program analyses and transformations. We believe that the further development of the abstract interpretation framework will make it possible to create new and integrated analyses that drive more powerful optimizations.

Mercury and HAL show that user supplied declarations for types, modes and determinism allow for much stronger program analysis results and hence more strongly optimized compilation. In our work we have borrowed from this experience by supporting a limited form of mode declarations for CHR and shown that such mode declarations allow for considerable optimizations in the case of ground constraints. In (Sneyers, Schrijvers, and Demoen 2005b) also type declarations have been added to CHR. We would like to further extend the support for these declarations and take their information into account in more analyses and optimizations.

The study of the union-find program has shown the importance of the constraint stores in the efficiency and time complexity of CHR programs. Our contribution is the use of hash tables for ground constraints. Earlier proposals for constraint store data structures by (Holzbaur, García de la Banda, Stuckey, and Duck 2005) were global variables and search trees. Both these data structures are used to speed up lookups based on equality constraints in the guard. It should be investigated whether good data structures exist to improve lookups based on other guard constraints, e.g. search trees for inequality constraints. In addition, the data structures should be composed in different ways. For example, an equality based lookup based on a variable and a ground term may best be served by a hash tabled stored in an attribute attached to the variable. A constraint should also move from one data structure to another as it becomes more and more instantiated.

Different Execution Strategies The theoretical operational semantics ω_t of CHR is a very high-level semantics with a considerable amount of non-determinism. The refined operational semantics ω_r is a particular instance of ω_t with less non-determinism. Both semantics are formulated in terms of sequential derivations.

While these two semantics impose certain restrictions on execution, there is still quite some freedom left. For example, two successive derivation steps that do

not involve the same constraints may be applied simultaneously. This can be the basis for a parallel execution strategy. Mozart seems a suitable implementation language as it allows for cheap and easy parallelism and distributed execution. Another variation on the ω_r semantics is the reordering of rules in case this does not affect final states of derivations. A confluence analysis could be used to establish the latter property and heuristics or profiling techniques could be used to select a good ordering of rules. Of course, the introduction of non-termination should be avoided.

In addition to automatic and preset execution strategies the non-determinism in the theoretical operational semantics may be left to the programmer to resolve. For example, the **Solve** transition leaves unspecified in what order constraints are pushed onto the execution stack. This may be fixed by a programmer-provided priority function. The effect of different priority functions would be similar to the effect of different propagator buffers in a propagator-based constraint solver (Schulte and Stuckey 2004). Another approach, from term rewriting systems (see Section 2.5.2), is to have the programmer express the execution strategy in terms of a number of strategy primitives.

Appendix A

Source Code: `wfs`

```
/*

        File:      wfs.chr
        Author:           Tom Schrijvers
        E-mail:           Tom.Schrijvers@cs.kuleuven.ac.be
        Copyright: 2003 - 2004, K.U.Leuven

        Computes well-founded semantics of logic program

*/

:- module(wfs,[prog/0]).

:- use_module(library(chr)).
:- use_module(library(lists)).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
constraints
                inactivate1/1,
                fire1/1,

                inIplus1/1,
                inIminus1/1,
                head1/2,
                inBodyPlus1/2,
                inBodyMinus1/2,
                literal1/2,
                pliteral1/2,
                headof1/2,
```

```
                fire2/1,

                inIplus2/1,
                head2/2,
                inBodyPlus2/2,
                pliteral2/2,

                undefined2/1,
                atmost/0,
                atmost_end/0,

                atleast/0,
                atleast_end/0,

                undefined/1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% atleast()

clean1_1 @
        inIplus1(P), head1(R,P) \ inBodyPlus1(_,R) <=>
                true.
clean1_2 @
        inIplus1(P), head1(R,P) \ inBodyMinus1(_,R) <=>
                true.
clean1_3 @
        inIplus1(P) \ headof1(P,_) <=>
                true.
clean1_4 @
        inIplus1(P) \ head1(R,P), literal1(R,_), pliteral1(R,_) <=>
                true.
clean1_5 @
        inIminus1(P), head1(R,P) \ inBodyPlus1(_,R) <=>
                true.
clean1_6 @
        inIminus1(P), head1(R,P) \ inBodyMinus1(_,R) <=>
                true.
clean1_7 @
        inIminus1(P) \ headof1(P,_) <=>
                true.
clean1_8 @
```

```
        inIminus1(P) \ head1(R,P), literal1(R,_), pliteral1(R,_) <=>
                true.

% fire1()

fire_posq @
        inIplus1(P) \ inBodyPlus1(P,R), pliteral1(R,NP) <=>
                fire1(R),
                 NP1 is NP - 1, pliteral1(R,NP1).
fire_negq @
        inIminus1(P) \ inBodyMinus1(P,R) <=>
                fire1(R).
fire @
        fire1(R), literal1(R,NU) <=>
                 NU1 is NU - 1, literal1(R,NU1).
all_literals_true @
        literal1(R,0), head1(R,P), pliteral1(R,0) <=>
                inIplus1(P).

% inactivate1()

inactivate1_posq @
        inIplus1(P) \ inBodyMinus1(P,R) <=>
                inactivate1(R).
inactivate1_negq @
        inIminus1(P) \ inBodyPlus1(P,R) <=>
                inactivate1(R).
inactivate1_clean1 @
        inactivate1(R) \ inBodyMinus1(_,R) <=>
                true.
inactivate1_clean2 @
        inactivate1(R) \ inBodyPlus1(_,R) <=>
                true.
inactivate1 @
        inactivate1(R), head1(R,PP), literal1(R,_),
        pliteral1(R,_), headof1(PP,N) <=>
                N1 is N - 1, headof1(PP,N1).
no_active_rules   @
        headof1(P,0) <=>
                inIminus1(P).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% atmost()

atmost, headof1(P,_)     ==> undefined2(P).
atmost, inBodyPlus1(P,R)  ==> inBodyPlus2(P,R).
atmost, head1(R,P)         ==> head2(R,P).
atmost, pliteral1(R,I)    ==> pliteral2(R,I).
atmost                         <=> atmost_end.

% cleaning

clean2_1 @
        inIplus2(At), head2(Cl,At) \ inBodyPlus2(_,Cl) <=>
                true.
clean2_2 @
        inIplus2(At) \ undefined2(At) <=>
        true.

% fire2()

fire_posq2 @
        inIplus2(P) \ inBodyPlus2(P,R) <=>
                fire2(R).
fire2 @
        fire2(R), pliteral2(R,I) <=>
                J is I - 1, pliteral2(R,J).
all_pos_literals_true @
        head2(R,P), pliteral2(R,0) <=>
                inIplus2(P).

% atmost_end

clean2_end_1 @
        atmost_end \ inBodyPlus2(_,_) <=>
                true.
clean2_end_2 @
        atmost_end \ head2(_,_)       <=>
                true.
clean2_end_3 @
        atmost_end \ pliteral2(_,_)   <=>
                true.
clean2_end_3 @
        atmost_end \ inIplus2(_)   <=>
```

```
                true.

if_change @
        undefined2(_) \ atmost_end <=>
                atleast.
if_no_change @
        atmost_end                  <=>
                atleast_end.

to_atleast @
        atleast \ undefined2(P) <=>
                inIminus1(P).
to_atmost @
        atleast <=>
                atmost.

% atleast_end

        atleast_end \ headof1(P,_) <=> undefined(P).

        atleast_end \ head1(_,_) <=>
                true.
        atleast_end \ inBodyPlus1(_,_) <=>
                true.
        atleast_end \ inBodyMinus1(_,_) <=>
                true.
        atleast_end \ literal1(_,_) <=>
                true.
        atleast_end \ pliteral1(_,_) <=>
                true.
        atleast_end <=>
                true.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Example Program

% a :- a.
% b :- b.
% b :- \+ a.
% c :- \+ b.
% c :- c.
```

```
prog :-
        headof1(a,1), inBodyPlus1(a,r1), inBodyMinus1(a,r3),
        head1(r1,a), literal1(r1,1), pliteral1(r1,1),
        headof1(b,2), inBodyPlus1(b,r2), inBodyMinus1(b, r4),
        head1(r2,b), literal1(r2,1), pliteral1(r2,1),
        head1(r3,b), literal1(r3,1), pliteral1(r3,0),
        headof1(c,2), inBodyPlus1(c,r5),
        head1(r4,c), literal1(r4,1), pliteral1(r4,0),
        head1(r5,c), literal1(r5,1), pliteral1(r5,1),
        atmost.
```

Appendix B

Prolog Benchmarks

Table B.1 lists the results for the traditional Prolog benchmarks on 5 different Prolog systems as a reference for the CHR benchmark results. All measurements have been made on an Intel Pentium 4 2.00 GHz with 512 MB of RAM. Timings are relative to SICStus. The Prolog systems used are SICStus 3.12.0 and Yap 4.4.4 on the one hand and hProlog 2.4.11-32, SWI-Prolog 5.5.8 and XSB 2.6.1 on the other hand.

Benchmark	SICStus		Yap	hProlog	SWI-Prolog	XSB
boyer	1,110	100.0%	85.6%	106.3%	382.9%	218.0%
browse	1,670	100.0%	76.0%	71.3%	419.8%	129.8%
cal	1,500	100.0%	97.3%	69.3%	267.9%	138.7%
chat	930	100.0%	80.6%	79.6%	213.9%	108.5%
crypt	1,850	100.0%	96.8%	52.4%	293.0%	138.3%
ham	1,160	100.0%	81.0%	94.8%	344.8%	127.6%
meta_qsort	1,090	100.0%	87.2%	106.4%	406.3%	193.6%
nrev	1,170	100.0%	44.4%	60.7%	767.5%	94.9%
poly_10	1,290	100.0%	97.7%	84.5%	277.4%	195.2%
queens10	2,130	100.0%	67.1%	74.6%	405.1%	136.2%
queens_16	2,410	100.0%	52.7%	58.5%	408.3%	149.4%
reducer	990	100.0%	74.7%	92.9%	338.3%	173.5%
send	1,560	100.0%	59.0%	64.7%	241.0%	90.4%
tak	1,170	100.0%	86.3%	82.9%	393.9%	147.8%
zebra	960	100.0%	63.5%	101.0%	217.6%	105.1%
average	-	100.0%	76.7%	80.0%	358.5%	143.1%

Table B.1: Runtime performance of 15 Prolog benchmarks in 5 different Prolog systems.

Bibliography

Abdennadher, S. 1997. Operational Semantics and Confluence of Constraint Propagation Rules. In *CP'97: Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, G. Smolka, Ed. Springer Verlag, Schloss Hagenberg, Austria, 252–266.

Abdennadher, S. and Frühwirth, T. 1998. On Completion of Constraint Handling Rules. In *CP'98: Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*. Springer Verlag, Pisa, Italy.

Abdennadher, S., Frühwirth, T., and Meuss, H. 1996. On confluence of constraint handling rules. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science. Springer Verlag, Cambridge, USA.

Abdennadher, S., Krämer, E., Saft, M., and Schmauss, M. 2001. JACK: A Java Constraint Kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming, Kiel*. Kiel, Germany.

Abdennadher, S. and Marte, M. 2000. University course timetabling using constraint handling rules. *Applied Artificial Intelligence 14,* 4, 311–325.

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.

Alberti, M., Chesani, F., Guerri, A., Gavanelli, M., Lamma, E., Mello, P., Milano, M., and Torroni, P. 2005. Expressing interaction in combinatorial auction through social integrity constraints. In *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, A. Wolf, T. Frühwirth, and M. Meister, Eds. Ulm, Germany, 53–64.

Alberti, M., Ciampolini, A., Gavanelli, M., Lamma, E., Mello, P., and Torroni, P. 2003. Logic Based Semantics for an Agent Communication Language. In *FAMAS 2003: In Proceedings of the International Workshop on Formal Approaches to Multi-Agent Systems*. Warsaw, Poland, 21–36.

Berman, K. A., Schlipf, J. S., and Franco, J. V. 1995. Computing well-founded semantics faster. In *Logic Programming and Non-monotonic Reasoning*. 113–126.

Brand, D., Darringer, J., and Joyner, W. 1978. Completeness of Conditional Reductions. Tech. Rep. RC7404, IBM Research Center, Yorktown Heigghts, New York, USA. December.

Brass, S., Dix, J., Freitag, B., and Zukowski, U. 1998. Transformation-Based Bottom-Up Computation of the Well-Founded Model. Tech. Rep. 15–98, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz.

Chen, W. and Warren, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM 43,* 1, 20–74.

Christiansen, H. 2002. Logical Grammars Based on Constraint Handling Rules. In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*, P. J. Stuckey, Ed. Springer Verlag, Copenhagen, Denmark, 481.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. 2003. The Maude 2.0 System. In *RTA 2003: Rewriting Techniques and Applications*, R. Nieuwenhuis, Ed. Number 2706 in Lecture Notes in Computer Science. Springer Verlag, 76–87.

Codish, M., Demoen, B., and Sagonas, K. 1998. Semantic-based Program Analysis for Logic-based Languages Using XSB. *International Journal of Software Tools for Technology Transfer 2,* 1 (January), 29–45.

Codish, M., Falaschi, M., Marriott, K., and Winsborough, W. H. 1993. Efficient Analysis of Concurrent Constraint Logic Programs. In *ICALP'93: Proceedings of the 20th International Colloquium on Automata, Languages and Programming*. Springer Verlag, London, UK, 633–644.

Codognet, C., Codognet, P., and Corsini, M. 1990. Abstract Interpretation for Concurrent Logic Languages. In *NACLP'90: Proceedings of the North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, MA, USA, 215–232.

Coquery, E. 2003. TCLP: A type checker for CLP($\mathcal{X}$). In *Proceedings of the 13th Workshop on Logic Programming Environments*, F. Mesnard and A. Serebrenik, Eds. Katholieke Universiteit Leuven, 17–30.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press.

Cousot, P. and Cousot, R. 1977. Abstract Interpretation: A Unifed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-*

SIGPLAN symposium on Principles of programming languages. ACM Press, Los Angeles, California, 238–252.

Cui, B. and Warren, D. S. 2000a. A System for Tabled Constraint Logic Programming. In *CL 2000: Proceedings of the 1st International Conference on Computational Logic*, J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer Verlag, London, UK, 478–492.

Cui, B. and Warren, D. S. 2000b. Attributed Variables in XSB. In *Electronic Notes in Theoretical Computer Science*, I. Dutra et al., Eds. Vol. 30. Elsevier.

Davis, R., Buchanan, B. G., and Shortliffe, E. H. 1984. Production Rules as a Representation for a Knowledge-Based Consultation Program. In *Readings in Medical Artificial Intelligence: The First Decade*, W. J. Clancey and E. H. Shortliffe, Eds. Addison-Wesley, Reading, MA, USA, 98–130.

Demoen, B. hProlog. http://www.cs.kuleuven.ac.be/ bmd/hProlog/.

Demoen, B. 2002. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium. October.

Demoen, B., García de la Banda, M., Harvey, W., Marriott, K., and Stuckey, P. J. 1999. An Overview of HAL. J. Jaffar, Ed. Lecture Notes in Computer Science, vol. 1713. Springer Verlag, Alexandria, Virginia, USA, 174–188.

Demoen, B. and Nguyen, P.-L. 2000. So many WAM variations, so little time. In *CL2000: Proceedings of the 1st International Conference on Computational Logic*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, Eds. LNAI, vol. 1861. ALP, Springer Verlag, Londong, UK, 1240–1254.

Dershowitz, N. 1993. A Taste of Rewriting. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, P. Lauer, Ed. International Lecture Series 1991–92. Springer Verlag, 199–228.

Diaz, D. and Codognet, P. 1993. A minimal extension of the wam for clp(fd). In *ICLP'93: Proceedings of the 10th International Conference on Logic Programming*. MIT Press, Budapest, Hungary, 774–790.

Du, X., Ramakrishnan, C. R., and Smolka, S. A. 2000. Tabled Resolution + Constraints: A Recipe for Model Checking Real-Time Systems. In *IEEE Real Time Systems Symposium*. Orlando, Florida.

Duck, G., Schrijvers, T., and Stuckey, P. 2004. Abstract Interpretation for Constraint Handling Rules. Report CW 391, K.U.Leuven, Department of Computer Science, Leuven, Belgium. September.

Duck, G. J., García de la Banda, M., and Stuckey, P. J. 2004. Compiling Ask Constraints. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 3132. Springer Verlag, St-Malo, France, 105–119.

Duck, G. J., Stuckey, P. J., García de la Banda, M., and Holzbaur, C. 2003. Extending arbitrary solvers with constraint handling rules. In *PPDP'03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*. ACM Press, Uppsala, Sweden, 79–90.

Duck, G. J., Stuckey, P. J., García de la Banda, M., and Holzbaur, C. 2004. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 3132. Springer Verlag, St-Malo, France, 90–104.

Ehrig, H. 1979. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*. Springer Verlag, London, UK, 1–69.

Forgy, C. L. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In *Artificial Intelligence*. Vol. 19. North Holland Conference, 17–37.

Friendman-Hill, E. 2003. *Jess in Action*. Manning Publications.

Frühwirth, T. 1993. Entailment Simplification and Constraint Constructors for User-Defined Constraints. In *WCLP'93: Proceedings of the 3rd Workshop on Constraint Logic Programming*. Marseille, France.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming 37,* 1–3 (October), 95–138.

Frühwirth, T. 2002a. As Time Goes By: Automatic Complexity Analysis of Concurrent Rule Programs. In *KR2002: Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning*. Toulouse, France.

Frühwirth, T. 2002b. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. In *Electronic Notes in Theoretical Computer Science*, A. D. Pierro and H. Wiklicky, Eds. Vol. 59. Elsevier.

Frühwirth, T. and Abdennadher, S. 2003. *Essentials of Constraint Programming*. Cognitive Technologies. Springer Verlag.

Frühwirth, T. and Brisset, P. 1998. Optimal Placement of Base Stations in Wireless Indoor Telecommunication. In *CP'98: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*,

M. J. Maher and J.-F. Puget, Eds. Lecture Notes in Computer Science, vol. 1520. Springer Verlag, Pisa, Italy, 476–480.

Frühwirth, T. and Holzbaur, C. 2003. Source-to-Source Transformation for a Class of Expressive Rules. In *AGP 2003: Joint Conference on Declarative Programming*, F. Buccafurri, Ed. Reggio Calabria, Italy, 386–397.

Frühwirth, T. W. 2000. Proving Termination of Constraint Solver Programs. In *New Trends in Constraints*, K. R. Apt, A. C. Kakas, E. Monfroy, and F. Rossi, Eds. Lecture Notes in Computer Science, vol. 1865. Springer Verlag, Paphos, Cyprus, 298–317.

Galil, Z. and Italiano, G. F. 1991. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys 23,* 3, 319–344.

Ganzinger, H. and McAllester, D. 2001. A new meta-complexity theorem for bottom-up logic programs. In *International Joint Conference on Automated Reasoning*. Lecture Notes in Computer Science 2083. Springer Verlag, 514–528.

Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J.-P. 1993. Introducing OBJ. In *Applications of Algebraic Specification using OBJ*, J. Goguen, Ed. Cambridge.

Goguen, J. A. and Malcolm, G. 1996. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA, USA.

Gorn, S. 1965. Explicit Definitions and Linguistic Dominoes. In *Systems and Compter Science*. London, Ontario, USA, 77–115.

Gosling, J., Joy, B., and Steele, G. L. 1996. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc.

Guo, H.-F. and Gupta, G. 2003. Simplifying Dynamic Programming via Tabling. In *Proceedings of CICLOPS 2003. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto*, R. Lopes and M. Ferreira, Eds. 21–32.

Holzbaur, C. 1992. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Tech. Rep. TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria.

Holzbaur, C. and Frühwirth, T. 1999. Compiling constraint handling rules into Prolog with attributed variables. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, G. Nadathur, Ed. Number 1702 in Lecture Notes in Computer Science. Springer Verlag, 117–133.

Holzbaur, C. and Frühwirth, T. 2000. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules 14,* 4 (April).

Holzbaur, C., García de la Banda, M., Stuckey, P. J., and Duck, G. J. 2005. Optimizing Compilation of Constraint Handling Rules in HAL. *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules 5*. To appear.

IC-Parc. ECL^{i}PS^e. http://www.icparc.ic.ac.uk/eclipse/.

ILOG. 2004. *JRules 4.6 Technical White Paper*.

Intelligent Systems Laboratory. 2003. *SICStus Prolog User's Manual*. PO Box 1263, SE-164 29 Kista, Sweden.

ISO/IEC. 1995. *Information technology—Programming languages—Prolog—Part 1: General core*. ISO/IEC 13211-1:1995.

Jaffar, J. and Maher, M. J. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. C. 1992. The CLP($\Re$) Language and System. *ACM Trans. Program. Lang. Syst. 14,* 3, 339–395.

Knuth, D. E. and Bendix, P. B. 1970. Simple Word Problems in Universal Algebra. In *Computational Problems in Abstract Algebra*. Pergamon Press, 263–297.

Lloyd, J. W. 1987. *Foundations of logic programming; (2nd extended ed.)*. Springer Verlag.

Marriott, K., Søndergaard, H., and Jones, N. D. 1994. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems 16,* 3, 607–648.

Marriott, K. and Stuckey, P. J. 1998. *Programming with Constraints: an Introduction*. MIT Press.

Meier, M. Sepia*: The constraint logic programming system. http://www.clps.de/.

Mukund, M., Ramakrishnan, C. R., Ramakrishnan, I. V., and Verma, R. 2000. Symbolic Bisimulation using Tabled Constraint Logic Programming. In *International Workshop on Tabulation in Parsing and Deduction*. Vigo, Spain.

Pemmasani, G., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2002. Efficient Model Checking of Real Time Systems Using Tabled Logic Programming and Constraints. In *International Conference on Logic Programming*. Lecture Notes in Computer Science. Springer, Copenhagen, Denmark.

Pugh, W. Proposal for java memory model and thread specification revision. JSR-133, http://www.jcp.org/en/jsr/detail?id=133.

Pugh, W. et al. 2004. JSR-133: Java^{TM} Memory Model and Thread Specification. Sent to Final Approval Ballot, http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf.

Rao, P., Sagonas, K. F., Swift, T., Warren, D. S., and Freire, J. 1997. XSB: A system for effciently computing WFS. In *Logic Programming and Non-monotonic Reasoning*. 431–441.

Sahlin, D. and Carlsson, M. 1991. Variable Shunting for the WAM. Tech. Rep. SICS/R-91/9107, SICS.

Santos Costa, V., Damas, L., Reis, R., and Azevedo, R. 2004. *YAP User's Manual*. http://www.ncc.up.pt/~vsc/Yap/.

Saraswat, V. 2004. Concurrent Constraint-based Memory Machines: A framework for Java Memory Models (Preliminary Report). Tech. rep., IBM. March.

Savely, R. et al. 2005. *CLIPS Reference Manual*.

Schrijvers, T. 2004. JmmSolve: a generative Java memory model implemented in Prolog and CHR. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer Verlag, St-Malo, France. Poster presentation.

Schrijvers, T. 2005. A Collection of Assorted CHR Benchmarks. http://www.cs.kuleuven.ac.be/~toms/Research/CHR/.

Schrijvers, T. and Demoen, B. 2004a. Antimonotony-based delay avoidance for CHR. Report CW 385, K.U.Leuven, Department of Computer Science. July.

Schrijvers, T. and Demoen, B. 2004b. The K.U.Leuven CHR system: Implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, T. Frühwirth and M. Meister, Eds. Ulm, Germany, 1–5.

Schrijvers, T., Demoen, B., Duck, G., Stuckey, P., and Frühwirth, T. 2005a. Automatic Implication Checking for CHR Constraint Solvers. Report CW 402, K.U.Leuven, Department of Computer Science, Leuven, Belgium. January.

Schrijvers, T., Demoen, B., Duck, G., Stuckey, P., and Frühwirth, T. 2005b. Automatic Implication Checking for CHR Constraints. In *RULE'05: Proceedings of the 6th International Workshop on Rule-Based Programming*, H. Cirstea and N. Martí-Oliet, Eds. Nara, Japan.

Schrijvers, T. and Frühwirth, T. 2004. Implementing and Analysing Union-Find in CHR. Report CW 389, K.U.Leuven, Department of Computer Science. July.

Schrijvers, T. and Frühwirth, T. 2005. Analysing the CHR implementation of union-find. In *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, A. Wolf, T. Frühwirth, and M. Meister, Eds. Ulm, Germany, 135–146.

Schrijvers, T. and Frühwirth, T. 2005. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*. Accepted.

Schrijvers, T., Stuckey, P., and Duck, G. 2005. Abstract Interpretation for Constraint Handling Rules. In *PPDP'05: Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming*. ACM Press, Lisbon, Portugal.

Schrijvers, T. and Warren, D. S. 2004. Constraint handling rules and tabled execution. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer Verlag, St-Malo, France, 120–136.

Schrijvers, T., Warren, D. S., and Demoen, B. 2003. CHR for XSB. In *CICLOPS 2003: Proceedings of the Colloquium on Implementation of Constraint and LOgic Programming Systems*, R. Lopes and M. Ferreira, Eds. University of Porto, Mumbai, India, 7–20.

Schrijvers, T., Wielemaker, J., and Demoen, B. 2005. Poster: Constraint Handling Rules for SWI-Prolog. In *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, A. Wolf, Ed. Ulm, Germany.

Schulte, C. 2000. Programming Deep Concurrent Constraint Combinators. In *PADL'00: 2nd International Workhop of Practical Aspects of Declarative Languages*, E. Pontelli and V. Santos Costa, Eds. Lecture Notes in Computer Science, vol. 1753. Springer Verlag, Boston, MA, USA, 215–229.

Schulte, C. and Stuckey, P. J. 2004. Speeding up constraint propagation. In *PPCP 2004: Proceedings of the 9th International Conference on Principles and Practices of Constraint Programming*, M. Wallace, Ed. Lecture Notes in Computer Science, vol. 3258. Springer Verlag, 619–633.

Simons, P. 2000. Extending and implementing the stable model semantics. Ph.D. thesis, Helsinki University of Technology, Helsinki, Finland. Research Report 58.

Smolka, G. 1995. The Oz Programming Model. In *Computer Science Today: Recent Trends and Developments*, J. van Leeuwen, Ed. Lecture Notes in Computer Science, vol. 1000. Springer Verlag, Berlin, Germany, 324–343.

Sneyers, J., Schrijvers, T., and Demoen, B. 2005a. Guard Reasoning for CHR Optimization. Report CW 411, K.U.Leuven, Department of Computer Science, Leuven, Belgium.

Sneyers, J., Schrijvers, T., and Demoen, B. 2005b. Guard simplification in CHR programs. In *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, A. Wolf, Ed. Ulm, Germany, 123–134.

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.

Steele, G. 1984. *Common LISP: The Language*. Digital Press.

Stuckey, P. J. and Sulzmann, M. 2005. A Theory of Overloading. *ACM Transations on Programming Languages and Systems*. To appear.

Tarjan, R. E. and van Leeuwen, J. 1984. Worst-case Analysis of Set Union Algorithms. *Journal of the ACM 31,* 2, 245–281.

Thielscher, M. 2005. FLUX: A Logic Programming Method for Reasoning Agents. *Theory and Practice of Logic Programming*.

Toman, D. 1996. Computing the Well-founded Semantics for Constraint Extensions of Datalog. In *Proceedings of CP'96 Workshop on Constraint Databases*. Number 1191 in Lecture Notes in Computer Science. Cambridge, MA, USA, 64–79.

Toman, D. 1997. Memoing Evaluation for Constraint Extensions of Datalog. *Constraints: An International Journal, Special Issue on Constraints and Databases 2,* 3/4 (December), 337–359.

Van Gelder, A., Ross, K., and Schlipf, J. S. 1991. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM 38,* 3, 619–649.

Visser, E. 2001. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01)*, A. Middeldorp, Ed. Lecture Notes in Computer Science, vol. 2051. Springer Verlag, 357–361.

Warren, D. H. 1983. An Abstract Prolog Instruction Set. Technical note, Artificial Intelligence Center, SRI International.

Warren, D. S. et al. 2005. The XSB Programmer's Manual: version 2.7, vols. 1 and 2. http://xsb.sf.net.

Wielemaker, J. 2004. SWI-Prolog release 5.4.0. http://www.swi-prolog.org/.

Wolf, A. 2001. Adaptive Constraint Handling with CHR in Java. In *CP'01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science 2239. Springer Verlag, 256.

Wolf, A. 2005. Intellingent Search Strategies Based on Apdative Constraint Handling Rules. *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules 5*. To appear.

List of Symbols

The following list describes the meaning of symbols that are frequently used throughout the text. Some symbols may in specific cases be used in a different context; in those cases their meaning is always explicitly mentioned.

$\bar{s} = \bar{t}$	p. 9	pairwise equality of sequences
$\textsc{Base}(\mathbf{P})$	p. 41	the Herbrand base of the program $\mathbf{P}$
$\Box$	p. 9	the empty sequence
$[\![\mathcal{P}]\!]$	p. 52	the logical theory represented by program $\mathcal{P}$
$\equiv$	p. 10	syntactic equality
$\forall F$	p. 52	the universal closure of formula F
$\mathcal{T}$	p. 14	the logical theory of a constraint domain
$\mathcal{V}$	p. 10	the set of variables
$\mathit{meaning}(\sigma)$	p. 52	the logical meaning of execution state σ
$\neg \cdot S$	p. 42	the set formed by taking the complement of each literal in S
$c\#i$	p. 19	a constraint c with identifier i
ω_r	p. 22	the refined operational semantics of CHR
ω_t	p. 18	the operational semantics of CHR
$+\!+$	p. 9	sequence concatenation
$\mathcal{P}$	p. 16	a CHR program
Prog	p. 17	the set of all possible CHR programs
$\bar{\exists}_A F$	p. 10	projection of F onto $vars(A)$

$\mathcal{H}$	p. 15	the Herbrand constraint theory
$\rightarrowtail$	p. 20	a mapping from an execution state to an execution state
$\rightarrowtail^*$	p. 20	the transitive closure of $\rightarrowtail$
σ	p. 19	an execution state
θ	p. 11	a substitution
$\mathbf{U}_P(I)$	p. 42	the greatest unfounded set of $\mathbf{P}$ with respect to I
$\uplus$	p. 9	multiset union
$vars(E)$	p. 10	the set of variables occurring in E
$E \sim F$	p. 11	E and F are variants
f/n	p. 10	the function or predicate symbol with functor f and arity n
H_k	p. 18	the sequence of kept head constraints of a rule
H_r	p. 18	the sequence of removed head constraints of a rule
I	p. 42	a partial interpretation
$\mathcal{D}_b$	p. 16	the logical theory of built-in constraints
D	p. 14	a constraint domain
$\mathbf{\Phi}_P$	p. 43	the Fitting operator
$\mathbf{P}$	p. 41	a general logic program
$\mathbf{W}_P$	p. 42	the transformation whose fixed point is the well-founded semantics
$\mathbf{W}_P^*$	p. 42	the well-founded semantics of program $\mathbf{P}$
$\mathbf{\Sigma_a}$	p.126	the domain of abstract execution states
α	p.126	an abstraction function
$\mathcal{AS}$	p.127	the abstract semantic function
$\mathbf{\Sigma}$	p.121	the domain of concrete execution states
γ	p.126	a concretization function
ω_d	p.120	the refined denotational semantics of CHR

$\mathcal{S}$	p.122	the semantic function of ω_d
$A \hookrightarrow B$	p.122	(signature of) a partial function from A to B
$pp(\sigma)$	p.121	the program point of σ

Index

List of Publications

Articles in international reviewed journals

1. T. Schrijvers, and T. Frühwirth, Optimal Union-Find in Constraint Handling Rules, Theory and Practice of Logic Programming, 2005, accepted.

2. T. Schrijvers, M. Garía de la Banda, B. Demoen and P. Stuckey, Improving PARMA Trailing, Theory and Practice of Logic Programming, 2005, accepted.

Contributions at international conferences, published in proceedings

1. T. Schrijvers, G. Duck, and P. Stuckey, Abstract Interpretation for Constraint Handling Rules, Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming, Lisbon, Portugal, 2005, accepted.

2. T. Schrijvers, and D. Warren, Constraint Handling Rules and Tabled Execution, Logic Programming, 20th International Conference, ICLP 2004, Proceedings (Demoen, B. and Lifschitz, V., eds.), vol 3132, LNCS, pp. 120-136, 2004.

3. T. Schrijvers, and A. Serebrenik, Improving Prolog Programs: Refactoring for Prolog, Logic Programming, 20th International Conference, ICLP 2004, Proceedings (Demoen, B. and Lifschitz, V., eds.), vol 3132, LNCS, pp. 58-72, 2004.

4. T. Schrijvers, and B. Demoen, Combining an Improvement to PARMA Trailing with Trailing Analysis, Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (Kirchner, C., ed.), pp. 88-98, 2002.

5. T. Schrijvers, M. Garca de la Banda, and B. Demoen, Trailing Analysis for HAL, Logic programming , 18th International Conference, Proceedings (Stuckey, P., ed.), vol 2401, LNCS, pp. 38-53, 2002.

Contributions at international workshops, published in proceedings

1. T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth, Automatic Implication Checking for CHR Constraints, Proceedings of 6th International Workshop on Rule-Based Programming, Nara, Japan (Cirstea, H. and Marti-Oliet, N., eds.), 2005.

2. J. Sneyers, T. Schrijvers, and B. Demoen, Guard Simplification in CHR programs, Proceedings of 19th Workshop on (Constraint) Logic Programming, Ulm, Germany (Wolf, A., ed.), 2005.

3. T. Schrijvers, and T. Frühwirth, Analysing the CHR Implementation of Union-Find, Proceedings of 19th Workshop on (Constraint) Logic Programming, Ulm, Germany (Wolf, A., ed.), 2005.

4. T. Schrijvers, and B. Demoen, The K.U.Leuven CHR System: Implementation and Application, First workshop on constraint handling rules: selected contributions (Frühwirth, T. and Meister, M., eds.), pp. 1-5, 2004.

5. T. Schrijvers, A. Serebrenik, and B. Demoen, Refactoring Prolog Code, INAP / WLP 2004. 13th International conference on applications of declarative programming and knowledge management and 18th workshop on logic programming. Proceedings. (Seipel, D. and Hanus, M. and Geske, U. and Bartenstein, O., eds.), pp. 115-126, 2004.

6. T. Schrijvers, D. Warren, and B. Demoen, CHR for XSB, Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic Programming Systems (Lopes, R. and Ferreira, M., eds.), pp. 7-20, 2003.

7. T. Schrijvers, Combining an Improvement to PARMA Trailing with Analysis in HAL, Proceedings of CICLOPS'2002, the Colloquium of Constraint and LOgic Programming Systems (Demoen, B., ed.), pp. 1-12, 2002.

Contributions at international conferences, not published or only as abstract

1. T. Schrijvers, J. Wielemaker, and B. Demoen, Constraint Handling Rules for SWI-Prolog, 19th Workshop on (Constraint) Logic Programming, W(C)LP

2005, Ulm, Germany, February 21-23, 2005.

2. T. Schrijvers, JmmSolve: a Generative Java Memory Model Implemented in Prolog and CHR, 20th International Conference on Logic Programming, ICLP 2004, Saint-Malo, France, September 6-10, 2004.

Technical reports

1. J. Sneyers, T. Schrijvers, and B. Demoen, Guard Reasoning for CHR Optimization, Department of Computer Science, K.U.Leuven, Report CW 411, Leuven, Belgium, May, 2005.

2. T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth, Automatic Implication Checking for CHR Constraint Solvers, K.U.Leuven, Department of Computer Science, Report CW 402, January, 2005.

3. J. Sneyers, T. Schrijvers, and B. Demoen, Guard Simplification in CHR programs, K.U.Leuven, Department of Computer Science, Report CW 396, November, 2004.

4. G. Duck, T. Schrijvers, and P. Stuckey, An Abstract Interpretation Framework for Constraint Handling Rules, Department of Computer Science, K.U.Leuven, Report CW 391, Leuven, Belgium, September, 2004.

5. T. Schrijvers, and T. Frühwirth, Implementing and Analysing Union-Find in CHR, K.U.Leuven, Department of Computer Science, Report CW 389, July, 2004.

6. T. Schrijvers, and B. Demoen, Antimonotony-based Delay Avoidance for CHR, K.U.Leuven, Department of Computer Science, Report CW 385, July, 2004.

7. T. Schrijvers, and B. Demoen, JmmSolve: a Generative Reference Implementation of CCM machines, Department of Computer Science, K.U.Leuven, Report CW 379, Leuven, Belgium, January, 2004.

8. T. Schrijvers, A. Serebrenik, and B. Demoen, Refactoring Prolog Programs, Department of Computer Science, K.U.Leuven, Report CW 373, Leuven, Belgium, November, 2003.

9. T. Schrijvers, and B. Demoen, Combining an Improvement to PARMA Trailing with Analysis in HAL, Department of Computer Science, K.U.Leuven, Report CW 338, Leuven, Belgium, April, 2002.

10. T. Schrijvers, and B. Demoen, An Improvement to PARMA Variable Trailing, Department of Computer Science, K.U.Leuven, Report CW 326, Leuven, Belgium, December, 2001.

Biography

Tom Schrijvers was born on the 10th of June 1978 in Leuven. He finished High School at the St-Romboutscollege in Mechelen in 1996. In 1999 he received a Bachelor's degree of Science in Engineering (*Kandidaat Burgerlijk Ingenieur*) and in 2001 a Master's degree of Science in Engineering in Computer Science (*Burgerlijk Ingenieur in de Computerwetenschappen*) from the Katholieke Universiteit Leuven. His master thesis with the title "A Critical Study of Additional OOP Concepts for Java" was supervised by Professor Eric Steegmans and co-authored by Roel Hertoghs.

In August 2001 he joined the DTAI research group and started working as a Ph.D. student under the supervision of Professor Bart Demoen, funded by a teaching assistant position of the K.U.Leuven. In 2002 he became a research assistant funded by the Fund for Scientific Research Flanders (*F.W.O. Vlaanderen*). He was a visiting scholar at the Monash University in 2001 and 2003, at the State University of Melbourne in 2003, at the University of Melbourne in 2004 and at the University of Ulm in 2004.

He is the co-author of 2 articles in an international journal and of 12 papers published at international conferences and workshops and he has received the "Best Technical Paper Award" twice at the International Conference on Logic Programming.

Analyses, Optimalisaties en Uitbreidingen van Constraint Handling Rules

Samenvatting

1 Inleiding

Constraint Handling Rules (CHR) is een programmeertaal die gebaseerd is op regels en meestal ingebed wordt in een andere programmeertaal. Het is eenvoudige en tegelijk toch erg krachtige taal die elementen combineert van Constraint (Logic) Programming (CLP) en termherschrijfsystemen. Aanvankelijk was CHR bedoeld voor de implementatie van constraint solvers voor planning en optimalisatie, maar momenteel wordt ze gebruikt voor een brede waaier aan toepassingen: verwerking van natuurlijke taal, type-inferentie, multi-agent-systemen,...

Er bestaan verschillende implementaties van CHR in Prolog, maar ook in Java en Haskell. Algemeen wordt de implementatie van Christian Holzbaur in Prolog beschouwd als de referentie-implementatie. Recentelijk werd de verfijnde operationele semantiek van CHR geformuleerd die de belangrijkste kenmerken van deze implementatie vat.

Doelstellingen In deze thesis leveren we bijdragen op het gebied van programma-analyses, programma-optimalisaties en uitbreidingen van CHR:

- In het verleden is er weinig aandacht besteed aan geoptimaliseerde compilatie van CHR. De referentie-implementatie past slechts een gering aantal programma-specifieke optimalisaties toe. Slechts recentelijk hebben Duck et al. (Holzbaur, García de la Banda, Stuckey, and Duck 2005) een aantal meer substantiële bijdragen geleverd aan geoptimaliseerde compilatie.

In deze thesis presenteren wij meerdere nieuwe optimalisaties: codespecialisatie voor ground constraints, anti-monotonische delay avoidance, constraint stores gebaseerd op hashtabellen en een nieuwe late storage-optimalisatie. Daarnaast tonen we formeel de correctheid van enkele optimalisaties aan, iets dat voordien nog niet gebeurd was. Onze belangrijkste bijdrage op het gebied van geoptimaliseerde compilatie is echter ons voorstel tot een meer systematische aanpak van programma-analyse.

- Het doel van programma-analyse is de afleiding van nuttige eigenschappen van een programma. In het verleden zijn verscheidene analyses geformuleerd om theoretische eigenschappen af te leiden. Deze eigenschappen zijn nuttig om het gedrag en de correctheid van CHR-programma's na te gaan. We passen deze analyses toe in een gevalstudie om hun relevantie na te gaan in de praktijk.

  Een belangrijke klasse van programma-analyses is deze die programma-optimalisaties mogelijk maakt. Tot dusver is hierover erg weinig gepubliceerd voor CHR. Meestal wordt er informeel en vrij vaag over gesproken. De reden hiervoor is dat er geen systematische aanpak bestond en de analyses van minder belang werden geacht dan de eigenlijke optimalisaties. Het gebrek aan nauwgezette documentatie maakt het echter moeilijk om na te gaan of ze correct zijn, hoe ze verbeterd kunnen worden en hoe ze samengesteld kunnen worden tot meer complexe analyses.

  Ons raamwerk voor abstracte interpretatie van CHR biedt hiervoor een oplossing. Abstracte interpretatie (Cousot and Cousot 1977) is een algemene techniek voor programma-analyse die niet gebonden is aan een bepaalde programmeertaal. Ze is formeel van aard en legt het verband tussen de analyse en de operationele semantiek van de taal. Vanwege haar formele aard biedt abstracte interpretatie een geschikte systematische aanpak van programma-analyse voor CHR. Het raamwerk laat toe om algemene technieken te gebruiken om de correctheid van analyses aan te tonen en om analyses samen te stellen. We formuleren ons raamwerk in functie van de verfijnde denotationele semantiek en illustreren het gebruik met twee instanties: een late storage-analyse en een groundness-analyse.

- De eerste uitbreiding van de expressiviteit van CHR betreft de integratie van CHR met een meer expressieve variant van Prolog: getabuleerde uitvoering (Chen and Warren 1996). Getabuleerde uitvoering vermijdt automatisch vele vormen van non-terminatie van Prolog en is ook nuttig voor automatische performantieverbeteringen door haar dynamisch hergebruik van vorige berekeningen. Doordat de programmeur ontlast is van deze aspecten, kunnen programma's meer declaratief, compact en eenvoudig geformuleerd worden.

  De tweede uitbreiding voegt automatisch de functionaliteit toe aan CHR constraint solvers om de implicatie van constraints na te gaan. Deze functio-

naliteit is belangrijk om relevante informatie te verkrijgen uit een constraint store en om complexe constraints op te bouwen uit primitieve constraints. We gebruiken onze techniek om CHR solvers modulair samen te stellen.

Naast deze belangrijke doelstellingen streven we ook naar een betere verspreiding van de CHR-taal en willen we het praktisch nut van de taal aantonen. Het eerste aspect realiseren we met een nieuw state-of-the-art CHR-systeem dat beschikbaar is in drie Prolog-systemen. Voor het tweede aspect illustreren we het gebruik van CHR in drie toepassingen en tonen we aan dat het union-find-algoritme in CHR geïmplementeerd kan worden met de beste gekende tijdscomplexiteit.

Overzicht In Sectie 2 behandelen we eerst kort de belangrijkste begrippen omtrent CHR en gerelateerde onderwerpen. Sectie 3 illustreert vervolgens het gebruik van CHR bij drie toepassingen. Daarna bespreken we in Sectie 4 drie belangrijke theoretische eigenschappen van CHR-programma's en bestuderen we het nut hiervan bij een praktisch programma.

Een korte samenvatting van de referentie-implementatie van CHR wordt gegeven in Sectie 5. Onze eigen CHR-implementatie, het K.U.Leuven CHR-systeem bespreken we vervolgens in Sectie 6. In Sectie 7 formuleren we dan een raamwerk voor de abstracte interpretatie van CHR-programma's. De integratie van CHR met getabuleerde uitvoering komt daarna aan bod in Sectie 8. Als laatste bijdrage presenteren we in Sectie 9 het automatisch toevoegen van implicatiefunctionaliteit aan CHR constraint solvers. In Sectie 10 formuleren we tenslotte de conclusie van deze thesis en noemen we nog enkele mogelijkheden voor toekomstig werk op.

2 Inleidende begrippen en achtergrond

Constraint Handling Rules is ontworpen in de context van Constraint Logic Programming met als bedoeling om de gemakkelijke ontwikkeling van nieuwe constraint solvers mogelijk te maken.

Logic Programming De data van logic programming zijn termen, opgebouwd uit functiesymbolen en variabelen. Atomen zijn opgebouwd uit predikaatsymbolen en termen. Een logisch programma bestaat uit een aantal regels, clauses, met als hoofd een atoom en als lichaam een conjunctie van atomen. Intuïtief is de betekenis van een clause dat het atoom in het hoofd waar is als de atomen in het lichaam waar zijn.

Twee atomen of twee termen kunnen gelijk gemaakt worden met behulp van variabelensubstituties, variabelen die vervangen worden door termen. Operationeel berekent een logisch programma voor een gegeven atoom, de query, de variabelensubstituatie waaronder dit atoom waar is.

Constraint Solving en Constraint Logic Programming Analoog aan atomen worden constraints opgebouwd uit constraintsymbolen en variabelen en waarden. Een constrainttheorie is een logische theorie waaruit afgeleid kan worden welke constraints waar zijn en welke niet. Een conjunctie van constraints is satisfieerbaar als er een waardetoekenning bestaat voor de variabelen in de constraints zodat de constraints waar zijn onder de constrainttheorie. Het is de bedoeling van een constraint solver om na te gaan of een conjunctie van constraints satisfieerbaar is. Een typische strategie van een constraint solver is het herschrijven van de constraints met behulp van de constrainttheorie.

In Constraint Logic Programming wordt Logic Programming aangevuld met constraints: constraints mogen opgenomen worden als atomen in de lichamen van clauses. Tijdens de uitvoering van een programma worden alle constraints behandeld door de constraint solver.

Constraint Handling Rules CHR is een taal ingebed in een gasttaal die een aantal ingebouwde constraints aanbiedt. CHR-constraints zijn, analoog aan atomen, opgebouwd uit constraintsymbolen en termen. CHR-programma's bestaan uit een aantal CHR-regels. Elke regel heeft een hoofd, een guard en een lichaam. Het hoofd (H_k, H_k) bestaat uit een sequentie van CHR-constraints, de guard (G) uit een sequentie van ingebouwde constraints en het lichaam (B) uit een sequentie van beide soorten constraints. Er zijn drie soorten CHR-regels, schematisch:

- de **simplificatie**regel: H_r `<=>` G `|` B`.`
- de **simpagatie**regel: $H_k \backslash H_r$ `<=>` G `|` B`.`
- de **propagatie**regel: H_k `==>` G `|` B`.`

Operationeel wordt er een verzameling van CHR-constraints, de constraint store, gemanipuleerd. Een regel wordt uitgevoerd op voorwaarde dat er constraints in de constraint store zitten die overeenkomen met de constraints in het hoofd van de regel. Voor deze constraints moet aan de guard voldaan zijn. Bij de uitvoering van de regel worden de constraints H_r uit de constraint store verwijderd. De constraints in het lichaam van de regel worden toegevoegd aan de constraint store. De propagatieregel wordt slechts één maal uitgevoerd met dezelfde CHR-constraints.

Er zijn twee verschillende operationele semantieken geformuleerd op de bovenstaande regelsemantiek. De eerste en oorspronkelijke operationele semantiek van CHR laat toe dat regels in een willekeurige volgorde uitgevoerd worden. Deze semantiek wordt daarom de theoretische operationele semantiek genoemd. De tweede, verfijnde operationele semantiek legt de volgorde van regeltoepassingen wel vast. De constraint store van links naar rechts geordend worden. De regels moeten met de constraints van links naar rechts in de constraint store in tekstuele volgorde geprobeerd worden. Deze verfijnde semantiek dient als basis voor alle grote CHR-systemen.

Verband met andere regelgebaseerde talen Hoewel CHR ontstaan is in de context van Constraint Logic Programming, vertoont het ook overeenkomsten met andere programmeertalen die gebaseerd zijn op regels. Zo komen de operationele aspecten van CHR sterk overeen met die van Production Rule Systems. Beide bestaan uit conditionele regels die termen (of constraints) toevoegen aan en verwijderen uit een werkgeheugen (constraint store). CHR kan echter overweg met de ingebouwde constraints van de gasttaal en het herbekijken van CHR constraints bij het verwerken van ingebouwde constraints is dan ook specifiek voor CHR. Production Rule Systems zijn in de eerste plaats bedoeld voor expertsystemen, terwijl CHR constraint solvers en meer algemene problemen als toepassingen heeft.

Termherschrijfsystemen (TRSs) vertonen sterke gelijkenissen met CHR op het theoretische vlak. Beide zijn onderbouwd met een logische theorie. Bij CHR is dit een constrainttheorie, terwijl dit bij TRSs een theorie van algebraïsche gelijkheden is. TRSs herschrijven termen op basis van de gelijkheden, terwijl CHR constraint stores herschrijft op basis van equivalenties. Een aantal theoretische eigenschappen van TRSs, zoals confluentie, zijn overgenomen in CHR. Daarbij moeten wel een aantal CHR-specifieke aspecten in rekening gebracht worden, zoals de propagatieregels die geen zin hebben bij TRSs.

3 Drie toepassingen van CHR

We illustreren het gebruik van CHR bij drie toepassingen: een implementatie van het klassieke union-find-algoritme, een raamwerk voor het testen van nieuwe geheugenmodellen voor Java een een implementatie van de well-founded semantiek voor algemene logische programma's.

Union-Find Het union-find-algoritme is een klassiek algoritme uit de jaren '70. Het betreft een gegevensstructuur die een aantal disjuncte verzamelingen voorstelt. Elke verzameling heeft een bepaald element dat de vertegenwoordiger van die verzameling is. Er zijn drie operaties gedefinieerd op deze datastructuur:

- Het aanmaken van een nieuwe singletonverzameling met een nieuw element dat niet voorkomt in één van de reeds aanwezige verzamelingen.
- De unie van twee verzamelingen waarvan elk een element gegeven is.
- Het opzoeken van de vertegenwoordiger van een verzameling waarvan een element gegeven is.

Een naïeve implementatie bestaat erin om de verzamelingen voor te stellen als bomen. De elementen van de verzameling zijn knopen in de overeenkomstige boom en de vertegenwoordiger is de wortel van de boom. Het aanmaken van een singletonverzameling komt overeen met het maken van nieuwe boom met als wortel

het gegeven element. De unie van twee verzamelingen wordt gerealiseerd door de wortel van de ene boom een kind te maken van de wortel van de andere boom. Het opzoeken van de vertegenwoordiger voor een element gebeurt door van de knoop die overeenkomt met het element te vertrekken. Als die knoop de wortel is, dan komt het element overeen met de vertegenwoordiger. Anders herhaalt de procedure zich voor de ouder van de knoop. De unie-operatie is gedefinieerd in termen van de zoek-operatie; men moet namelijk eerst de wortels van de twee bomen zoeken. De zoek-operatie domineert dan ook de complexiteit. Ze is afhankelijk van de lengte d van het pad van het gegeven element tot de wortel van de boom. In het slechtste geval is dit n voor n elementen en de tijdscomplexiteit van een operatie is dan ook $\mathcal{O}(n)$.

Twee verbeteringen kunnen aan dit naïeve algoritme aangebracht worden. De eerste betreft een heuristiek om de unie-operatie te verbeteren. Als voor elke boom de diepte wordt bijgehouden, dan moet men ervoor zorgen de wortel van de diepste boom als wortel van de nieuwe boom te behouden. Op deze wijze probeert men de diepte van de bomen laag te houden. De tweede verbetering past de structuur van de boom aan tijdens de zoekoperatie. Elke knoop die men tegenkomt tijdens het zoeken naar de wortel maakt men een kindknoop van de wortel. Op deze wijze wordt de boom minder diep en zullen volgende zoekoperaties sneller de wortel vinden. Afzonderlijk verbeteren beide operaties niets aan de complexiteit, maar gezamenlijk zorgen ze ervoor dat in het slechtste geval de tijdscomplexiteit quasi-lineair is.

Onze CHR-implementaties van beide varianten van het algoritme zijn even compact als de oorspronkelijke imperatieve formuleringen en eenvoudig te begrijpen. De verbeterde versie vereist slechts lichte aanpassingen aan de naïeve versie.

Java Memory Model Een geheugenmodel van een programmeertaal specificeert de interactie tussen verschillende uitvoeringsthreads en het centrale geheugen. Het centrale geheugen bestaat uit geheugenlocaties die waarden bevatten. Een uitvoeringsthread kan op verschillende manieren met deze geheugenlocaties interageren: een leesoperatie leest de waarde in een locatie, een schrijfoperatie overschrijft de aanwezige waarde met een nieuwe waarde en een lockoperatie verhindert de interactie van andere uitvoeringsthreads met de locatie en een unlockoperatie heft het effect van een lockoperatie op. Het komt erop neer dat het geheugenmodel voor elke leesoperatie aangeeft door welke schrijfoperatie de gelezen waarde geschreven is.

Het oude geheugenmodel van Java bezat een aantal ongewenste eigenschappen en vertoonde onverwacht gedrag. Daarom werd er een oproep gedaan om een nieuw geheugenmodel met betere karakteristieken voor Java op te stellen.

Vijay Saraswat stelt een raamwerk voor, genaamd Concurrent Constraint-based Memory Machines (CCMMs), waarbinnen verschillende geheugenmodellen geformuleerd en bestudeerd kunnen worden. CCMMs reduceert een programma

tot een aantal operaties die inwerken op bepaalde variabelen in bepaalde threads. Een concreet geheugenmodel wordt dan declaratief geformuleerd in termen van een aantal regels die beschrijven hoe de operaties onderling geordend moeten worden en hoe lees- en schrijfoperaties aan elkaar gekoppeld moeten worden.

Om wille van de declaratieve aard van CCMMs hebben we ervoor gekozen om het raamwerk in een Constraint Logic Programming-taal te implementeren. Concreet hebben we SWI-Prolog met het K.U.Leuven CHR-systeem gebruikt. De ordeningsrelatie en de koppeling van waarden die berekend worden met behulp van rekenkundige operaties in het programma gebeurt met behulp van CHR constraint solvers. SWI-Prolog zelf is handig om de bronprogramma's te verwerken en de essentiële operaties te extraheren.

Het is erg makkelijk om regels aan te passen in onze implementatie en na te gaan wat de impact daarvan is op concrete testprogramma's. Helaas werd het CCMMs raamwerk slechts enkele maanden voor het verstrijken van de uiteindelijke beslissingsdatum voor het nieuwe geheugenmodel geformuleerd en kon daardoor niet voldoende interesse meer opwekken. Het door Java gekozen geheugenmodel is strikt imperatief van aard en werd veel eerder geformuleerd door vooraanstaande onderzoekers in de Java-gemeenschap.

Well-Founded Semantiek De well-founded semantiek is een semantiek voor algemene logische programma's die werd voorgesteld door Van Gelder, Ross en Schlipf (Van Gelder, Ross, and Schlipf 1991). Het is de bedoeling dat deze semantiek natuurlijk en intuïtief is. Een algemeen logisch programma bestaat uit Horn-clauses waarin negatieve doelen kunnen voorkomen.

Een interpretatie is een verzameling van positieve en negatieve doelen. De well-founded semantiek is geformuleerd als een transformatie van een interpretatie. Men vertrekt van de lege interpretatie en past herhaaldelijk twee verschillende transformaties toe tot de interpretatie niet meer verandert. De bekomen interpretatie bevat de doelen die waar en niet waar zijn in de well-founded semantiek.

Wij baseren ons op een van de vele algoritmes voor de berekening van de semantiek, een vereenvoudigde versie van een algoritme dat de stabiele semantiek van een logisch programma berekent. Het algoritme is zoals de semantiek zelf geformuleerd als een vastepuntsberekening van twee alternerende stappen. Elk van beide stappen bestaat uit een aantal operaties waarvan de onderlinge volgorde geen rol speelt. CHR is een geschikte taal om deze onderlinge vrijheid uit te drukken. De standaard operationele semantiek van CHR doet namelijk geen uitspraak over de volgorde. De operaties kunnen daarom dan ook erg compact geformuleerd worden. Voor de afwisseling van de twee uitvoeringsstappen en de vastepuntsberekening maken we gebruik van bindcode in CHR die wel rekening houdt met uitvoeringsvolgorde en de verfijnde operationele semantiek van CHR. Dit levert ons een programma op dat enkel aandacht besteedt aan uitvoeringsvolgorde waar het echt nodig is en op de andere plaatsen sterk vereenvoudigd is ten

opzichte van een imperatieve formulering.

4 Theoretische eigenschappen van CHR

We bestuderen de relevantie van drie theoretische eigenschappen van CHR-programma's. Deze eigenschappen zijn: de declaratieve semantiek van CHR, confluentie en tijdscomplexiteit. In de eerste plaats zijn deze eigenschappen nuttig om na te gaan of een CHR-programma de gewenste karakteristieken heeft.

We gaan na wat elk van deze eigenschappen ons leert over onze implementatie van het union-find-algoritme.

Declaratieve semantiek De declaratieve semantiek van een CHR-programma associeert een programma met een logische theorie. Deze theorie is opgebouwd uit een aantal logische formules, axioma's, één voor elke regel in het programma.

Onder deze declaratieve semantiek is het mogelijk om een logische betekenis toe te kennen aan de uitvoeringstoestand van de operationele semantiek van CHR. Deze logische betekenis is de conjunctie van alle ingebouwde en CHR-constraints die voorkomen in de uitvoeringstoestand.

Onder de logische semantiek herschrijft de operationele semantiek de initiële uitvoeringstoestand tot een finale uitvoeringstoestand. Daarbij hebben alle tussenliggende toestanden, inclusief de eerste en de laatste, dezelfde logische betekenis.

De declaratieve semantiek van een programma houdt niet altijd steek, maar is voornamelijk nuttig als de programmeur de intentie heeft om een bepaalde logische theorie (deels) te implementeren. Typisch zal de logische theorie van een programma niet equivalent zijn met de bedoelde logische theorie. Als de programmatheorie een deel is van de bedoelde theorie, dan zijn de resultaten bekomen met het programma voldoende. Dat wil zeggen dat elke eindtoestand dezelfde logische betekenis heeft als de initiële toestand, zowel onder de programmatheorie als onder de bedoelde theorie. Een noodzakelijk verband tussen beide theorieën heeft geen operationele relevantie.

Het union-find-algoritme is een implementatie van een gelijkheidsrelatie. Studie van de logische theorie van onze union-find-implementatie toont aan dat ze equivalent is met de logische theorie van gelijkheid. Dit komt duidelijk overeen met onze intentie. Elke bekomen eindtoestand drukt dus logisch dezelfde gelijkheden uit als de begintoestand.

Confluentie Een eindig CHR-programma is confluent als voor elke begintoestand alle mogelijke verschillende derivaties dezelfde eindtoestand opleveren.

Ondanks het grote indeterminisme in de operationele semantiek van CHR, garandeert confluentie toch een unieke eindtoestand. Dit is meestal een gewenste eigenschap die de programmeur toelaat zich niets aan te trekken van de concrete uitvoeringsstrategie van een bepaalde implementatie van CHR.

Het is mogelijk om confluentie na te gaan met behulp van een techniek ontleend aan termherschrijfsystemen. Men stelt alle kritische paren op, minimale toestanden waarop twee verschillende CHR-regels van toepassing zijn. Als deze kritische paren allen voor de twee verschillende derivaties dezelfde eindtoestand opleveren, dan is het programma confluent. Anders is het niet confluent.

Gerelateerde eigenschappen zijn canoniciteit en aanvulling. Een canonisch programma is een programma dat voor elke begintoestand die logisch equivalent is onder de programmatheorie dezelfde eindtoestand oplevert. Dit is duidelijk een nog sterkere eigenschap. Aanvulling is een techniek om een niet-confluent programma om te vormen tot een confluent programma door het aan te vullen met nieuwe regels.

Een studie van confluentie bij onze naïeve union-find-implementatie levert verschillende problematische kritische paren op. We kunnen deze paren naar oorzaak indelen in drie groepen. Een eerste groep is te wijten aan de inherente non-confluentie van het algoritme. De unie-operatie veroorzaakt een destructieve aanpassing van de toestand. Een zoek-operatie voor of na een unie-operatie levert dus een ander resultaat op. Een tweede groep van problematische paren veronderstelt toestanden die niet bereikbaar zijn bij een correct gebruik van het programma. Zo is het niet toegelaten om een element meer dan eens aan te maken. Als dit toch gebeurt, is het niet verwonderlijk dat het programma eigenaardig gedrag vertoont. De derde groep van paren geeft aan dat ons programma zich enkel goed gedraagt onder een strikt sequentiële uitvoering. Het afhandelen van een nieuwe constraint uitstellen levert een ongewenst resultaat op. Een studie van het optimale programma levert gelijkaardige resultaten op, behalve dan dat het aantal kritische paren een grote-orde talrijker is.

Tijdscomplexiteit De belangrijkst algemene resultaten op het gebied van tijdscomplexiteit in CHR zijn (Frühwirth 2002a; Frühwirth 2002b). Deze resultaten stellen een algemene formule voor met parameters die afhangen van het programma en het doel. Deze formule levert in de praktijk een vrij ruwe bovengrens op voor de tijdscomplexiteit van het programma. Daarom passen we een alternatieve techniek toe om de tijdscomplexiteit van onze union-find-implementaties te bepalen.

We tonen namelijk eerst met behulp van de verfijnde operationele semantiek de operationele equivalentie aan van onze CHR-implementaties met de overeenkomstige imperatieve implementaties met een gekende tijdscomplexiteit. Vervolgens tonen we aan dat de individuele uitvoeringsstappen in CHR en imperatieve talen dezelfde tijdscomplexiteit hebben. Hiervoor redeneren we over de implementatietechnieken van CHR en de hashtabellen die we zelf geïntroduceerd hebben voor de efficiënte implementatie van constraint stores. Hieruit kunnen we besluiten dat onze CHR-implementaties dezelfde tijdscomplexiteit hebben als hun imperatieve tegenhangers. Daarmee hebben we aangetoond dat het mogelijk is om het union-find-algoritme met de beste gekende tijdscomplexiteit te implementeren.

5 Implementatie van CHR

De CHR-implementatie in Prolog van Christian Holzbaur wordt algemeen beschouwd als de referentie-implementatie van CHR.

We beschrijven deze referentie-implementatie in termen van een vereenvoudigd compilatieschema. Dit vereenvoudigde compilatieschema sluit nauw aan bij de verfijnde operationele semantiek van Prolog. We komen dan tot de referentie-implementatie door een aantal optimalisaties en specialisaties toe te passen op het vereenvoudigde schema.

Naast de uitleg over het implementatieschema, geven we ook als eerste een formeel correctheidsbewijs voor optimalisaties. We baseren onze bewijzen op de verfijnde operationele semantiek.

6 Het K.U.Leuven CHR-systeem

Sinds de creatie van CHR zijn er een aantal verschillende implementaties van de taal gemaakt. Veel van deze implementaties zijn echter in onbruik geraakt. Bij de aanvang van deze thesis werd de implementatie in Prolog van Christian Holzbaur (Holzbaur and Frühwirth 1999) als de referentie-implementatie beschouwd. Daarnaast waren er enkele implementaties in Java en Haskell. Een belangrijk probleem bij al deze systemen is dat ze reeds geruime tijd niet meer aangepast zijn aan de nieuwste ontwikkelingen.

Bij de aanvang van deze thesis kwam er echter nieuwe interesse voor de implementatie van CHR en voor geoptimaliseerde compilatie in het bijzonder. In HAL werd een nieuw CHR-systeem geïmplementeerd (Holzbaur, García de la Banda, Stuckey, and Duck 2005). Ook wij hebben een nieuw CHR-systeem ontwikkeld, het K.U.Leuven CHR-systeem.

Met het K.U.Leuven CHR-systeem bieden we een degelijk alternatief in Prolog voor de referentie-implementatie. Het systeem implementeert de huidige stand van zaken op het gebied van geoptimaliseerde implementatie in een Prolog-implementatie. Het K.U.Leuven CHR-systeem implementeert met name vele van de optimalisaties uit (Holzbaur, García de la Banda, Stuckey, and Duck 2005): verbeterde join ordering, late storage, never stored constraints, continuatie-optimalisaties, ... Naast de optimalisaties uit ander werk, implementeert het systeem ook onze eigen bijdragen op het gebied van geoptimaliseerde compilatie: codespecialisatie voor ground constraints, hashtabellen als constraint stores en de anti-monotonische delay avoidance.

Oorspronkelijk was het K.U.Leuven CHR-systeem geschreven in hProlog. We hebben het systeem echter ook overgezet naar XSB en SWI-Prolog. Omdat het systeem voor een groot stuk gebaseerd is op de ISO Prolog-standaard en gebruik maakt van slechts een klein aantal systeem-specifieke functies, verloopt dit overzetten relatief eenvoudig.

Experimentele evaluatie toont aan dat onze individuele bijdragen op het gebied van geoptimaliseerde compilatie goede performantieverbeteringen opleveren. Bovendien is het systeem op zich een competitief systeem dat vergelijkbare of betere resultaten oplevert dan de referentie-implementatie.

7 Abstract interpretatie voor CHR

Hoewel CHR reeds geruime tijd bestaat en een degelijke referentie-implementatie heeft, was het aantal mensen betrokken bij de geoptimaliseerde compilatie van CHR tot de komst van nieuwe CHR-systemen (Holzbaur, García de la Banda, Stuckey, and Duck 2005; Schrijvers and Demoen 2004b) erg beperkt. De nood aan communicatie en vergelijking tussen CHR-systemen heeft intussen geleid tot de formulering van de verfijnde operationele semantiek van CHR (Duck, Stuckey, García de la Banda, and Holzbaur 2004).

Naast een formulering van de geïmplementeerde operationele semantiek, is er echter ook nood aan een formalisatie van programma-analyses voor CHR. De meeste beschikbare analyses zijn niet of eerder vaag omschreven en formele correctheidsbewijzen zijn niet beschikbaar.

Abstracte interpretatie is een algemene methodologie voor programma-analyse: een analyse wordt geformuleerd als een abstractie van het gewone uitvoeringsproces. Abstracte interpretatie biedt een oplossing voor de huidige moeilijkheden bij de correcte analyse van CHR-programma's en maakt het CHR-compilers mogelijk om complexere analyses te realiseren.

Wij stellen voor de algemene methodologie van abstracte interpretatie toe te passen op CHR. We formuleren daarom een raamwerk voor de abstracte interpretatie in termen van een verfijnde denotationele semantiek. Deze semantiek is een alternatieve formulering van de verfijnde operationele semantiek die nauwer aansluit bij de doeltalen van de CHR-compilatie: ze is gebaseerd op de notie van code-oproepen. De formulering van dit raamwerk in termen van de semantiek van CHR is niet triviaal door het grote niet-deterministische gehalte van CHR.

We illustreren het gebruik van ons raamwerk met twee analyses. De eerste analyse is een late storage analyse. Dit is een bestaande analyse voor CHR die we geformaliseerd hebben binnen ons raamwerk. Het doel van de analyse is na te gaan waar de overhead van het opslaan en verwijderen van een constraint in de constraint store vermeden kan worden. De tweede analyse is een groundness-analyse. Een groundness-analyse is een welbekende analyse in Prolog die uitzoekt welke variabelen tijdens de uitvoering volledig geïnstantieerd zijn. Onze groundness-analyse voor CHR is geparametriseerd in een groundness-analyse voor Prolog en het is makkelijk om één Prolog-analyse te vervangen door een andere. De informatie die afgeleid wordt door de groundness-analyse kan gebruikt worden om algemene code achterwege te laten die niet van toepassing is op ground constraints.

Deze twee analyses werden geïmplementeerd in het K.U.Leuven CHR-systeem

en geëvalueerd op een aantal benchmarks. Het blijkt dat reeds goede resultaten behaald worden, ondanks de eenvoud van de formulering van de analyses.

8 Integratie van CHR met getabuleerde uitvoering

XSB is een Prolog-systeem dat uitgebreid is met getabuleerde uitvoering. Getabuleerde uitvoering hergebruikt reeds uitgevoerde berekeningen en voorkomt hierdoor vele gevallen van non-terminatie. Vele toepassingen hebben baat bij tabulatie: parsers, programma-analyses, datamining, ...

In het verleden is er reeds herhaaldelijk behoefte geweest aan de combinatie van constraints en tabulatie. Dit was steeds een ingewikkelde onderneming. Initieel bood XSB helemaal geen ondersteuning voor constraints en moesten constraints gerealiseerd worden door koppelingen met externe bibliotheken van andere programmeertalen of door meta-vertolkers. Later kwam er primitieve ondersteuning voor constraints onder de vorm van geattribueerde variabelen (Cui and Warren 2000a).

We stellen voor om CHR te gebruiken om in XSB op een eenvoudige en expressieve wijze constraint solvers te realiseren. Dat maakt het mogelijk om in constrainttoepassingen te genieten van de gecombineerde expressiviteit van tabulatie en CHR.

Ons voorstel houdt heel wat implementatie-uitdagingen in. We hebben het K.U.Leuven CHR-systeem overgezet naar XSB. Hieraan hebben we dan de nodige ondersteuning toegevoegd voor tabulatie. Onze aanpak loopt gelijk aan die van (Cui and Warren 2000a): we formuleren een aantal nodige operaties voor de interactie tussen de CHR constraint store en de getabuleerde oproepen en antwoordtabellen. Met behulp van programmatransformatie worden deze operaties verwerkt in de getabuleerde predikaten. De programmeur wordt echter niet betrokken in de concrete implementatie van deze operaties; hij moet slechts via een aantal declaraties enkele keuzes aangeven.

We illustreren de combinatie van constraints en tabulatie op een klein transportprobleem. Het blijkt dat de combinatie nuttig is om de performantie van deze toepassing te verbeteren en dat de opties die we aanbieden uitgebuit kunnen worden om de performantie nog verder te verbeteren.

9 Automatische implicatietesten

Constraint solvers geschreven in CHR gaan de satisfieerbaarheid van een conjunctie van constraints na. De conjunctie wordt als doel opgegeven in de initiële uitvoeringstoestand. Als de uitvoering dan eindigt in een succestoestand, is de conjunctie (mogelijk) satisfieerbaar. Anders is ze zeker niet satisfieerbaar.

Buiten conjunctie zijn er echter ook andere interessante logische operatoren die nuttig zijn in de context van constraints. Met name implicatie kan gebruikt worden om uit een constraint store de gewenste informatie te halen. Als men geïnteressseerd is in een bepaalde eigenschap, gaat men na of deze geïmpliceerd wordt door de constraint store. Implicatie is ook een belangrijke bouwsteen voor de compositie van primitieve constraints tot complexe constraints. Vele CLP-systemen ondersteunen implicatie dan ook in één of andere vorm: constraintcombinatoren in Mozart (Schulte 2000), gereïfieerde constraints in de `clp(FD)`-bibliotheek en het predikaat `entailed/2` in de `clp(QR)` bibliotheek van SICStus Prolog (Intelligent Systems Laboratory 2003).

Wij breiden CHR tevens uit om automatisch implicaties na te gaan. In tegenstelling tot Prolog is dit niet triviaal. In Prolog moet men slechts een predikaat oproepen en als deze oproep slaagt, wordt het predikaat geïmpliceerd door het Prolog-programma. In CHR daarentegen wordt een opgeroepen constraint toegevoegd aan de conjunctie van constraints en krijgen we enkel een antwoord betreffende de satisfieerbaarheid hiervan.

We vergelijken twee verschillende technieken om implicatie na te gaan. De eerste techniek is de *kopieertechniek* en de tweede is de *trailingtechniek*. De kopieertechniek is gebaseerd op een vergelijking van de eindtoestanden van twee derivaties. De ene eindtoestand is voor het oorspronkelijke doel en de andere eindtoestand is voor de conjunctie van het oorspronkelijke doel met de constraint waarvan men de implicatie wil nagaan. Indien beide eindtoestanden gelijk zijn, dan is aan de implicatie voldaan. Bij de trailingtechniek wordt de constraint toegevoegd aan de eindtoestand van het oorspronkelijke doel. Tijdens de derivatie naar een nieuwe eindtoestand wordt een log bijgehouden van alle wijzigingen. Wijzigingen die elkaar opheffen worden automatisch geschrapt. Als het log op het einde leeg is, dan is aan de implicatie voldaan. We realiseren deze trailingtechniek door een automatische programmatransformatie die het oorspronkelijke CHR-programma omzet in een ander CHR-programma.

Beide technieken zijn voldoende, maar niet noodzakelijke testen voor implicatie. Als ze aangeven dat aan de implicatie voldaan is, dan is er zeker aan voldaan. Als ze echter niet aangeven dat aan de implicatie voldaan is, dan zou hier toch aan voldaan kunnen zijn. We bewijzen dat het zowel nodige als voldoende technieken zijn voor een bepaalde klasse van constraint solvers, namelijk canonieke solvers. Bovendien stellen we vast dat de technieken ook voldoende zijn voor enkele individuele constraint solvers die buiten deze klasse vallen.

We illustreren het nut van implicatie door het te gebruiken bij de constructie van modulaire hierarchieën van CHR solvers. De solvers zijn modulair doordat ze enkel met elkaar communiceren via een vaste interface en niets moeten afweten van de concrete implementaties van deze interfaces. Communicatie tussen solvers gebeurt in het bijzonder via guards. We laten toe dat de ene solver in zijn guards de implicatie nagaat van constraints in de andere solver. De andere solver laat op

zijn beurt weten aan de eerste solver wanneer het nuttig is om bepaalde implicaties opnieuw na te gaan via een zogenaamd event-mechanisme.

Een evaluatie van beide technieken geeft aan dat in een eenvoudige implementatie de trailingtechniek beter is dan de kopieertechniek. Bij een geoptimaliseerde implementatie zijn beide technieken echter aan elkaar gewaagd. Beide hebben weliswaar verschillende zwakke punten. De kopieertechniek gedraagt zich slechter als de te vergelijken constraint stores groot zijn, terwijl de trailingtechniek zich slechter gedraagt als de derivaties lang zijn.

De trailingtechniek heeft het voordeel dat ze makkelijk te realiseren is in een CHR-systeem. Ze is gebaseerd op een transformatie van het CHR-programma. De kopieertechniek daarentegen vereist een gedetailleerde kennis van de datastructuren die gebruikt zijn voor de constraint stores.

10 Besluit

Het doel van deze thesis was de studie van verscheidene aspecten van CHR die gerelateerd zijn aan programma-analyse, geoptimaliseerde compilatie en uitbreidingen van de expressiviteit.

We hebben onze studie uitgevoerd op basis van de implementatie van een nieuw CHR-systeem, het K.U.Leuven CHR-systeem. Verscheidene nieuwe optimalisaties en een algemeen raamwerk voor programma-analyse op basis van abstracte interpretatie werden geformuleerd en gevalideerd binnen dit CHR-systeem. De expressiviteit van CHR werd verbeterd door de integratie met getabuleerde uitvoering en door de automatische toevoeging van functionaliteit om implicaties na te gaan bij CHR constraint solvers.

Open problemen In de loop van deze thesis is het duidelijk geworden dat er vele uitdagingen blijven om CHR verder te ontwikkelen. De uitdagingen gerelateerd aan deze thesis liggen op twee gebieden: de bruikbaarheid van de taal en de efficiëntie van CHR-implementaties.

Op het gebied van bruikbaarheid vermelden we een viertal open problemen:

- In deze thesis hebben we aangetoond dat verschillende analyses van theoretische eigenschappen een beperkte bruikbaarheid hebben in de praktijk. Het is daarom een belangrijk open probleem om nieuwe eigenschappen en analyses uit te denken die nuttig zijn om de correctheid van praktische implementaties na te gaan.

- CHR biedt momenteel geen ondersteuning voor hergebruik van codefragmenten en veel gebruikte programmeeridiomen. Degelijke functionaliteit voor hergebruik zal het mogelijk maken om kwalitatief hoogstaande code te hergebruiken en veel sneller nieuwe CHR-programma's te schrijven.

- CHR is in de eerste plaats ontworpen om constraint solvers in te schrijven. Toch bevat de taal hiervoor slechts basisondersteuning. In deze thesis hebben we reeds bijkomende ondersteuning toegevoegd, namelijk voor het automatisch nagaan van implicaties. In samenwerking met Peter Stuckey hebben we reeds onderzoek gestart naar automatische ondersteuning voor constraint-projectie.

  Onze ondersteuning voor implicaties in CHR constraint solvers is een eerste stap in de richting van communicatie tussen constraint solvers. De toevoeging van prioriteiten voor constraints en constraint solvers, zoals dit bijvoorbeeld mogelijk is in ECL^{i}PS^e (IC-Parc), zou het mogelijk moeten maken om de programmeur meer inspraak te geven in de wisselwerking tussen constraint solvers.

- Momenteel is CHR enkel beschikbaar in een beperkt aantal talen. Het meest aanwezig is CHR in Prolog, met name in SICStus, ECL^{i}PS^e, Yap, SWI-Prolog en XSB. Momenteel worden de nodige voorbereidingen getroffen om het K.U.Leuven CHR-systeem ook in Ciao Prolog op te nemen. Met deze dekking van Prolog-systemen is CHR beschikbaar voor bijna alle Prolog-programmeurs.

  Prolog is echter geen erg populaire taal als het op het aantal programmeurs en industriële toepassingen aankomt. Hoewel een paar CHR-systemen bestaan voor de meer populaire object-georiënteerde taal Java, zijn deze niet voorzien van de laatste nieuwe verbeteringen en richten zij zich niet op een vlotte integratie met Java.

  Om CHR makkelijker beschikbaar te kunnen maken in een groot aantal programmeertalen met de laatste nieuwe verbeteringen op implementatiegebied, hebben we met Christian Holzbaur het project opgevat om een CHR compiler te ontwikkelen die code genereert in een intermediair formaat. Deze intermediaire code kan dan omgezet worden naar alle gewenste doeltalen. Alle verbeteringen in de CHR compiler op het gebied van de intermediaire code kunnen zo onmiddellijk hun weg vinden naar de doeltalen.

Op het gebied van efficiëntie zien we de volgende twee grote uitdagingen:

- Een verbetering van de schaalbaarheid is van essentieel belang om CHR geschikt te maken voor praktische toepassingen waarin grote hoeveelheden gegevens verwerkt moeten worden.

  In deze thesis hebben we reeds een aantal optimalisaties geformuleerd en we zijn van mening dat de verdere ontwikkeling van het raamwerk voor abstracte interpretatie het mogelijk zal maken om nieuwe en samengestelde analyses te formuleren die meer krachtige optimalisaties mogelijk maken.

Van de programmeertalen Mercury en HAL leren we dat declaraties voor modes en determinisme veel sterkere resultaten opleveren voor programma-analyse en bijgevolg ook voor geoptimaliseerde compilatie. In ons werk hebben we deze ervaring toegepast door een beperkte vorm van zulke declaraties voor CHR mogelijk te maken. In het bijzonder hebben we aangetoond dat geoptimaliseerde compilatie zeer goede resultaten oplevert voor constraints die als ground gedeclareerd zijn. We zouden de ondersteuning van deze declaraties willen uitbreiden en ze gebruiken in meer analyses en optimalisaties.

De studie van het union-find-programma heeft het belang van efficiënte constraint stores aangetoond voor de efficiëntie en tijdscomplexiteit van CHR-programma's. Onze bijdrage is het gebruik van hashtabellen als constraint stores. In het verleden zijn er reeds andere datastructuren voorgesteld, zoals globale variabelen en zoekbomen. Verder onderzoek naar geschikte datastructuren is aangewezen.

- De theoretische operationele semantiek van CHR is erg vaag; ze biedt erg veel keuzevrijheid. De verfijnde operationele semantiek beperkt deze keuzevrijheid in zekere mate. Beide semantieken zijn geformuleerd in termen van sequentiële afleidingen.

  Hoewel beide semantieken bepaalde beperkingen opleggen aan de uitvoering, is er nog een grote mate aan keuzevrijheid. Bijvoorbeeld, twee opeenvolgende afleidingsstappen die geen gemeenschappelijke constraints beschouwen, kunnen ook gelijktijdig toegepast worden. Dit kan als basis dienen voor een parallelle uitvoeringsstrategie. De programmeertaal Mozart lijkt een geschikte kandidaat om deze parallelle strategie in te implementeren, want ze biedt goede ondersteuning voor zowel parallellisme als gedistribueerde uitvoering.

  Een andere variatie op de verfijnde operationele semantiek zou de herordening van CHR regels zijn, als deze geen impact hebben op de eindtoestand van de afleiding. Een confluentie-analyse kan gebruikt worden om deze eigenschap vast te stellen, terwijl heuristieken en metingen tijdens de uitvoering kunnen aangeven welke een goede ordening van regels is. Natuurlijk moet er hierbij op gelet worden dat er geen non-terminatie geïntroduceerd wordt.

  Naast deze automatische en voorgedefinieerde uitvoeringsstrategieën, is het ook mogelijk om de open keuzes door de programmeur te laten beslissen. Bijvoorbeeld de volgorde waarin constraints gereactiveerd worden kan door de programmeur bepaald worden op basis van prioriteiten. Het effect van verschillende prioriteiten zou hetzelfde kunnen zijn als dat van verschillende propagatorbuffers in constraint solvers gebaseerd op propagators (Schulte and Stuckey 2004).