Constraint Handling Rules – Compilation, Execution, and Analysis

Editors: Thom Frühwirth and Frank Raiser

Prof. Dr. Thom Frühwirth
Ulm University
Institute of Software Engineering and Programming Languages
Faculty of Engineering, Computer Science and Psychology
89069 Ulm, Germany
http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/

Dr. Frank Raiser
89075 Ulm, Germany
http://frankraiser.de

What are Constraint Handling Rules?

"one of the most powerful multiset rewriting languages"

— Professor Kazunori Ueda and Norio Kato, Waseda University, Japan, in 'Programming Logical Links'

"a powerful, highly optimized, lazy rule engine [...] consistently outperforms Rete-based systems"

— Peter Van Weert, K.U. Leuven, Belgium, in 'Efficient Lazy Evaluation of Rule-Based Programs'

"has the potential to become a lingua franca, a hub which collects and dispenses research efforts from and to the various related fields"

— Jon Sneyers, K.U. Leuven, Belgium, in 'Optimizing Compilation and Complexity of CHR'

"perfectly suitable for high level design of constraint systems"

— Marco Alberti and Evelina Lamma, Universita degli Studi di Ferrara, Italy, in 'Merging Views into CSPs: an Application for Computer Vision'

Constraint Handling Rules (CHR) is both a versatile theoretical formalism based on logic and an efficient practical high-level programming language based on rules and constraints.

Procedural knowledge is often expressed by if-then rules, events and actions are related by reaction rules, change is expressed by update rules. Algorithms are often specified using inference rules, rewrite rules, transition rules, sequents, proof rules, or logical axioms. All these kinds of rules can be almost directly written in CHR.

The clean logical semantics of CHR facilitates non-trivial program analysis and transformation. About a dozen free implementations of CHR exist in Prolog, Haskell, Javascript, Java, and C. CHR is also available online. More than 200 academic and industrial projects worldwide use CHR, more than 200 books and 2000 research papers reference it.

This book accompanies the monography Constraint Handling Rules by Thom Frühwirth, Cambridge University Press, 2009. Visit `http://www.constraint-handling-rules.org` for much more on CHR.

The Chinese letter comprising the CHR logo is transliterated as "CHR" and stems from the sign for "horse". It means to be fast, to be famous.

Foreword

Constraint Handling Rules (CHR) is a high-level programming language based on multi-headed, committed-choice, guarded multiset rewrite rules. The CHR language has been actively developed for over 20 years now and has become a major declarative specification and implementation language for constraint-based algorithms and applications.

Research in the CHR community has been fostered through numerous meetings throughout the past years, including seven workshops and a summer school specifically on CHR, as well as countless research visits.

Apart from increased international collaboration, new theoretical results and optimized implementations, this has led to many more CHR users and researchers. The recently published book on Constraint Handling Rules by Thom Frühwirth and this book complement each other: The former is a thorough introduction to all aspects of CHR, whereas this book presents recent research in implementation, extensions, and novel analyses of CHR.

In order to be self-contained, it starts with an introduction to CHR, which in the spirit of this book, is held concise and research-oriented. After that, carefully selected chapters from recent PhD theses provide detailed information on the topics compilation and optimization, execution strategies, and formal analysis of CHR. These chapters can be read individually based on the reader's interest.

The chapters have been edited by Thom Frühwirth and Frank Raiser to better suit the book's general theme. Additionally, the book has been reviewed by the individual authors of the chapters, the editors, and Florian Geiselhart and Johannes Langbein. The involved PhD theses range from 2005 up to the latest theses available at the time of writing, resulting in the following list of authors: Gregory J. Duck, Leslie De Koninck, Edmund S. L. Lam, Frank Raiser, Tom Schrijvers, and Jon Sneyers.

March 2011, Ulm — Thom Frühwirth and Frank Raiser

I am happy to present the slightly revised large print edition of this research book.

January 2018, Ulm — Thom Frühwirth

Contents

Part I

Introduction to CHR

Chapter 1

Constraint Handling Rules

Author:	Jon Sneyers
Thesis Title:	Optimizing Compilation and Computational Complexity of Constraint Handling Rules
School:	K.U.Leuven, Belgium
Publication Year:	2008

In this chapter we give an introduction to the CHR language, in terms of both theory and practice — that is, the practice of programming in CHR. The next chapter covers the practice of CHR from the perspective of the implementation of CHR compilers.

For other, more gentle or more complete introductions to CHR we refer to any of the following: [Frühwirth, 1998], [Frühwirth and Abdennadher, 2003], [Schrijvers, 2005], [Duck, 2005], [Sneyers et al., 2010b], and [Frühwirth, 2009]. Readers that are already familiar with CHR can skip this chapter, except for Section 1.1.6 which introduces some new terminology. All other material in this chapter consists of the usual definitions as in the above literature.

1.1 Syntax and semantics of CHR

In this section we introduce the syntax and semantics of Constraint Handling Rules. We assume the reader to be familiar with some basic notions of contraint logic programming.

1.1.1 History

Thom Frühwirth [1992] designed a new special-purpose programming language (or rather, a language *extension*), called *Constraint Handling Rules*. From its conception, CHR was meant to be used as a high-level and declarative language for implementing constraint solvers. As the name "Constraint Handling Rules" suggests, CHR programs consist of rules to handle user-defined constraints. The rules eventually reduce the user-defined constraints to built-in constraints. CHR is a language *extension*: it adds new functionality to an existing programming language, which is called the *host language* and which provides the built-in constraints. Historically, CHR was usually added to Prolog, but today there are also several CHR systems in other languages, like Java and Haskell.

As time went by, it became clear that CHR and its variants can be used for reasoning systems in general (not just constraint solvers), including deduction and abduction. In CHR, techniques like forward and backward chaining, bottom-up and top-down evaluation,

integrity constraints, tabling and memoization can easily be implemented and combined [Holzbaur and Frühwirth, 2000a].

More recently, CHR is seen as a concurrent very-high-level general-purpose programming language, especially suitable for rapid prototyping. Because CHR is such a high-level language, CHR programs are often more concise than pseudo-code descriptions of the corresponding algorithms.

The growing scope of CHR — from a special-purpose language for implementing constraint solvers to a general-purpose language — was made possible by increasingly more efficient CHR systems. In turn, improving the performance of general-purpose CHR programs (for instance, a CHR implementation of the classic union-find algorithm) motivated the creation of more efficient CHR systems.

1.1.2 Syntax

First of all, it should be clear that CHR is not meant to be used as a stand-alone programming language in itself. Instead, it is a language *extension* that adds user-defined constraints and rules to handle them to some given *host language*. We always assume that CHR is embedded in a host language $\mathcal{H}$ that provides data types and a number of predefined constraints. These predefined constraints are called host language constraints or *built-in* constraints. The traditional host language of CHR is Prolog. Its only host language constraint is equality of Herbrand terms (solved using Prolog's built-in unification); its data types are Prolog variables and terms. We denote the host language in which CHR is embedded between round brackets: i.e. CHR($\mathcal{H}$) denotes CHR embedded in host language $\mathcal{H}$. Most systems are CHR(Prolog) systems, but there are also several implementations of CHR(Java) and CHR(Haskell), and recently a CHR(C) system was developed. We discuss these CHR systems in Section 1.3.

Host language requirements

We assume the host language to offer at least one data type that can be used as an identifier, i.e. a data type that allows the following two operations: creation of a new unique value, and equality testing; for instance Prolog variables or integers. We denote the built-in constraint theory by $\mathcal{D}_{\mathcal{H}}$ and we assume that it defines at least the basic constraints `true`, the empty constraint which is trivially satisfied; `fail`, a contradictory constraint which is unsatisfiable; and the *ask*-versions of syntactic equality ("`==`") and inequality ("`\==`").

Syntax of CHR

CHR constraint symbols are drawn from the set of predicate symbols, denoted by a functor/arity pair (the functor is the name of the predicate, the arity is the number of arguments). CHR constraints, also called constraint atoms, user-defined constraints, or constraints for short, are atoms constructed from these symbols and the data types provided by the host language. A *query* or *goal* is a conjunction (or multiset, in an abstract setting) of both CHR and host language constraints. We denote the set of goals for a given CHR program $\mathcal{P}$ and host language $\mathcal{H}$ by the symbol $\mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$.

A CHR program $\mathcal{P}$ consists of a sequence of CHR rules. There are three kinds of rules: (where $k, l, m, n \geq 1$)

- Simplification rules: $h_1, \ldots, h_n \iff g_1, \ldots, g_m \mid b_1, \ldots, b_k.$
- Propagation rules: $h_1, \ldots, h_n \implies g_1, \ldots, g_m \mid b_1, \ldots, b_k.$
- Simpagation rules: $h_1, \ldots, h_l \setminus h_{l+1}, \ldots, h_n \iff g_1, \ldots, g_m \mid b_1, \ldots, b_k.$

The sequence, or conjunction, $h_1, \ldots, h_n$ consists of CHR constraints; together they are called the *head* or *head constraints* of the rule. A rule with n head constraints is called an *n-headed* rule and when $n > 1$, it is a *multi-headed* rule.

All the head constraints of a simplification rule and the head constraints $h_{l+1}, \ldots, h_n$ of a simpagation rule are called *removed* head constraints. The other head constraints — all heads of a propagation rule and $h_1, \ldots, h_l$ of a simpagation rule — are called *kept* head constraints. An *occurrence number* is associated with every head constraint. Head constraints are numbered per functor/arity pair, starting from 1, from the first rule to the last rule, removed heads before kept heads, from left to right.

The conjunction $b_1, \ldots, b_k$ consists of CHR constraints and host language constraints; it is called the *body* of the rule. The part of the rule between the arrow and the body is called the *guard*. It is a conjunction of host language constraints. The guard "$g_1, \ldots, g_m \mid$" is optional; if omitted, it is considered to be "`true` $\mid$".

A rule is optionally preceded by *name* @ where *name* is a term. No two rules may have the same name. If a rule does not have a name, it gets a unique name implicitly.

For simplicity, both simplification and propagation rules are often treated as special cases of simpagation rules. The following notation is used, where H^i is a sequence of CHR constraints, and G and B are guard and body as defined above:

$$H^k \setminus H^r \iff G \mid B$$

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. We use H_i to denote the heads of the i-th rule of a program.

1.1.3 Semantics: informal introduction by example

A derivation starts from an initial query: a multiset of constraint atoms, given by the user. This multiset of constraints is called the *constraint store*. The derivation proceeds by applying the rules of the program, which modify the constraint store. When no more rules can be applied, the derivation ends; the final constraint store is called the solution or solved form.

Rules modify the constraint store in the following way. A simplification rule can be considered as a rewrite rule which replaces the left-hand side (the head constraints) with the right-hand side (the body constraints), on the condition that the guard holds. The double arrow indicates that the head is logically equivalent to the body, which justifies the replacement. The intention is that the body is a simpler, or more canonical form of the head.

In propagation rules, the body is a consequence of the head: given the head, the body may be added (if the guard holds). Logically, the body is implied by the head so it is redundant. However, adding redundant constraints may allow simplifications later on. Simpagation rules are a hybrid between simplification rules and propagation rules: the constraints before the backslash are kept, while the constraints after the backslash are removed.

Listing 1.1: LEQ: Solver for the less-than-or-equal constraint

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

Example 1.1.1 (less-than-or-equal)**.** *The CHR program* LEQ *is shown in Listing 1.1. It is a classic CHR program to solve less-than-or-equal constraints. The first rule, reflexivity, replaces the trivial constraint* `leq(X,X)` *by* `true`*. Operationally, this entails removing this constraint from the constraint store (the multiset of all known CHR constraints). The second rule, antisymmetry, states that* `leq`(x,y) *and* `leq`(y,x) *are logically equivalent to* $x = y$*. Operationally this means that constraints matching the left-hand side may be removed from the store, after which the Prolog built-in equality constraint solver is used to unify* x *and* y*. The third rule, idempotence, removes redundant copies of the same* `leq`*/2 constraint. It is necessary to do this explicitly since CHR has a multiset semantics. The last rule, transitivity, is a propagation rule that computes the transitive closure of the* `leq`*/2 relation.*

An example derivation for the LEQ program would be the following:

		`leq(A,B), leq(B,C), leq(C,A)`
(transitivity)	$\rightarrowtail$	`leq(A,B), leq(B,C), leq(C,A), leq(A,C)`
(antisymmetry)	$\rightarrowtail$	`leq(A,B), leq(B,C), A=C`
(Prolog)	$\rightarrowtail$	`leq(A,B), leq(B,A), A=C`
(antisymmetry)	$\rightarrowtail$	`A=B, A=C`

Starting from the same initial query, multiple derivations may be possible. For example, another derivation is the following:

		`leq(A,B), leq(B,C), leq(C,A)`
(transitivity)	$\rightarrowtail$	`leq(A,B), leq(B,C), leq(C,A), leq(B,A)`
(antisymmetry)	$\rightarrowtail$	`leq(B,C), leq(C,A), A=B`
(Prolog)	$\rightarrowtail$	`leq(A,C), leq(C,A), A=B`
(antisymmetry)	$\rightarrowtail$	`A=C, A=B`

In the case of the LEQ program, it can be shown that all derivations ultimately lead to the same result. If a program has this property, we say it is *confluent*. Confluence will be discussed in Section 1.2.

Example 1.1.2 (prime numbers)**.** *Listing 1.2 shows a simple CHR(Prolog) program called* PRIMES*, a CHR variant of the Sieve of Eratosthenes. Dating back to 1992 [Frühwirth, 1992], this is one of the very first examples where CHR is used as a general-purpose programming language. Given a query of the form "*`upto`(n)*", where* n *is a positive integer, it computes all prime numbers up to* n*. The first rule (loop) does the following: if* $n > 1$*, it simplifies* `upto`(n) *to* `upto`$(n-1)$ *and adds a* `prime`(n) *constraint. The second rule handles the case for* $n = 1$*, removing any* `upto`*(1) constraint. Removing a constraint is done by simplifying it to the built-in constraint* `true`*. The third and most interesting rule (absorb) is a simpagation rule. If there are two* `prime`*/1 constraints* `prime`(a) *and*

Listing 1.2: PRIMES: Prime number generator

```
loop   @ upto(N) <=> N>1 | prime(N), upto(N-1).
stop   @ upto(1) <=> true.
absorb @ prime(A) \ prime(B) <=> B mod A =:= 0 | true.
```

`prime`*(b), such that b is a multiple of a, the latter constraint is removed. The effect of the absorb rule is that all non-primes are eventually removed. As a result, if the rules are applied exhaustively, the remaining constraints correspond exactly to the prime numbers up to n.*

1.1.4 Logical semantics

There are two ways to define the meaning of CHR programs (or declarative programming languages in general): a *logical* semantics formulates the meaning of a program in terms of a mapping to logical theories, while an *operational* semantics describes the behavior of a program, usually in terms of a state transition system that models program execution. Of course there should be a correspondence between both kinds of semantics. Soundness and completeness results, linking the logical semantics and the operational semantics, can be found in [Frühwirth, 1998].

In this section we describe the two main logical semantics of CHR: the original *classical* logic semantics, and the recent *linear* logic semantics, which was motivated by the shift towards general-purpose programming.

Classical logic semantics

Let $\bar{x}$ denote the variables occurring only in the body of the rule. A simplification rule $H \iff G \mid B$ corresponds to a logical equivalence, under the condition that the guard is satisfied: $\forall(G \rightarrow (H \leftrightarrow \exists\bar{x}B))$. A propagation rule $H \implies G \mid B$ corresponds to a logical implication if the guard is satisfied: $\forall(G \rightarrow (H \rightarrow \exists\bar{x}B))$. A simpagation rule $H^k \setminus H^r \iff G \mid B$ corresponds to a conditional equivalence: $\forall(G \rightarrow (H^k \rightarrow (H^r \leftrightarrow \exists\bar{x}B)))$. The (classical) logical semantics [Frühwirth, 1998] of a CHR program — also called its logical reading, declarative semantics, or declarative interpretation — is given by the built-in constraint theory $\mathcal{D}_\mathcal{H}$ (which defines the built-ins of the host language $\mathcal{H}$) in conjunction with the logical formulas for each rule. As an example, consider again the program LEQ of Example 1.1.1. The logical formulas corresponding to its rules are the following:

$$\begin{cases} \forall x, y : x = y \rightarrow (\texttt{leq}(x, y) \leftrightarrow \texttt{true}) & \textit{(reflex.)} \\ \forall x, y, x', y' : x = x' \wedge y = y' \rightarrow (\texttt{leq}(x, y) \wedge \texttt{leq}(y', x') \leftrightarrow x = y) & \textit{(antisym.)} \\ \forall x, y, x', y' : x = x' \wedge y = y' \rightarrow (\texttt{leq}(x, y) \rightarrow (\texttt{leq}(x', y') \leftrightarrow \texttt{true})) & \textit{(idempot.)} \\ \forall x, y, y', z : y = y' \rightarrow (\texttt{leq}(x, y) \wedge \texttt{leq}(y', z) \rightarrow \texttt{leq}(x, z)) & \textit{(transit.)} \end{cases}$$

or equivalently:

$$\begin{cases} \forall x : \texttt{leq}(x, x) & \textit{(reflexivity)} \\ \forall x, y : \texttt{leq}(x, y) \wedge \texttt{leq}(y, x) \leftrightarrow x = y & \textit{(antisymmetry)} \\ \texttt{true} & \textit{(idempotence)} \\ \forall x, y, z : \texttt{leq}(x, y) \wedge \texttt{leq}(y, z) \rightarrow \texttt{leq}(x, z) & \textit{(transitivity)} \end{cases}$$

Note the strong correspondence between the syntax of the CHR rules, their logical reading, and the natural definition of partial order.

The classical logical reading, however, does not reflect CHR's multiset semantics (the *idempotence* rule is logically equivalent to *true*). Also, the classical logic reading does not always make sense. For example, consider the classical logic reading of the PRIMES program of Example 1.1.2

$$\begin{cases} \forall n : n > 1 \rightarrow \texttt{upto}(n) \leftrightarrow \exists n' \texttt{prime}(n) \wedge n' = n - 1 \wedge \texttt{upto}(n') & \textit{(loop)} \\ \texttt{upto}(1) \leftrightarrow \texttt{true} & \textit{(stop)} \\ \forall a, b : b \ \texttt{mod} \ a = 0 \rightarrow \texttt{prime}(a) \rightarrow (\texttt{prime}(b) \leftrightarrow \texttt{true}) & \textit{(absorb)} \end{cases}$$

which is equivalent to:

$$\begin{cases} \forall n > 1 : \texttt{upto}(n) \leftrightarrow \texttt{prime}(n) \wedge \texttt{upto}(n-1) & \textit{(loop)} \\ \texttt{upto}(1) & \textit{(stop)} \\ \forall a, b : \texttt{prime}(a) \wedge b \ \texttt{mod} \ a = 0 \rightarrow \texttt{prime}(b) & \textit{(absorb)} \end{cases}$$

The last formula nonsensically states that a number is prime if it has a prime factor.

Linear logic semantics

For general-purpose CHR programs such as PRIMES, or programs that rely on CHR's multiset semantics, the classical logic reading is often inconsistent with the intended meaning. To overcome these limitations, Bouissou [2004] and Betz and Frühwirth [2005, 2007] independently proposed an alternative declarative semantics based on (intuitionistic) linear logic. The latter, most comprehensive study provides strong soundness and completeness results, as well as a semantics for the CHR$^\vee$ extension of CHR (see Section 1.6.1). For CHR programs whose constraints represent a multiset of resources, or whose rules represent unidirectional actions or updates, a linear logic semantics proves much more adequate. A simple example is the following coin-throwing simulator (which depends on the nondeterminism in the operational semantics):

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

The classical logic reading of this program entails $\texttt{head} = \texttt{tail}$. The linear logic reading of the coin-throwing program boils down to the following formula:

$$!(\texttt{throw}(\mathit{Coin}) \multimap (\mathit{Coin} = \texttt{head}) \& (\mathit{Coin} = \texttt{tail}))$$

In natural language, this formula means "you can always replace $\texttt{throw}(\mathit{Coin})$ with either $(\mathit{Coin} = \texttt{head})$ or $(\mathit{Coin} = \texttt{tail})$, but not both". This corresponds to the committed-choice and unidirectional rule application of CHR.

1.1.5 Abstract operational semantics ω_t

In this section we present the (abstract) operational semantics ω_t of CHR, sometimes also called *theoretical* or *high-level* operational semantics.

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.

1. Solve. $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$
where c is a built-in constraint and $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset} \mathbb{B}$.

2. Introduce. $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
where c is a CHR constraint and $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset} \mathbb{B}$.

3. Apply. $\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle C \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
where $\mathcal{P}$ has a (renamed apart) rule of the form $r \ @ \ H_1' \setminus H_2' \iff G \mid C$,
θ is a matching substitution with $chr(H_1) = \theta(H_1')$ and $chr(H_2) = \theta(H_2')$,
$h = (r, id(H_1), id(H_2)) \notin \mathbb{T}$, and $\mathcal{D}_{\mathcal{H}} \models (\bar{\exists}_{\emptyset} \mathbb{B}) \wedge (\mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G))$.

Figure 1.1: Transitions of the abstract (theoretical) operational semantics ω_t

Definition 1.1.3 (identified CHR constraint)**.** *An identified CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i, called 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, e.g. $id(\mathbb{S}) = \{i | c\#i \in \mathbb{S}\}$. Note that $chr(\mathbb{S})$ is a multiset although $\mathbb{S}$ is a set.*

Definition 1.1.4 (execution state)**.** *An execution state σ is a tuple $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The goal $\mathbb{G} \in \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ is a multiset of constraints to be rewritten to solved form. The CHR constraint store $\mathbb{S}$ is a set of identified CHR constraints that can be matched with rules in the program $\mathcal{P}$. The built-in constraint store $\mathbb{B}$ is the conjunction of all built-in constraints that have been passed to the underlying solver. This abstracts the internal representation used by the host language. Its actual meaning depends on the host language $\mathcal{H}$. The propagation history $\mathbb{T}$ is a set of tuples, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. It is used 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 $n \in \mathbb{N}$ represents the next free integer that can be used to identify a CHR constraint.*

We use $\sigma, \sigma_0, \sigma_1, \ldots$ to denote execution states and Σ^{CHR} to denote the set of all states. Transitions are defined by the binary relation $\rightarrowtail_{\mathcal{P}}$: $\Sigma^{\text{CHR}} \rightarrow \Sigma^{\text{CHR}}$ shown in Figure 1.1. Execution proceeds by exhaustively applying the transitions, starting from an initial state. We define $\rightarrowtail_{\mathcal{P}}^*$ as the transitive closure of $\rightarrowtail_{\mathcal{P}}$.

1.1.6 Derivations

Definition 1.1.5 (initial state)**.** *Given an initial goal (query) $\mathbb{G} \in \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$, we define $initstate(\mathbb{G}) = \langle \mathbb{G}, \emptyset, \texttt{true}, \emptyset \rangle_1$. The set of initial states is denoted by Σ^{init}.*

Definition 1.1.6 (final state)**.** *A final state $\sigma_f = \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ is an execution state for which no transition applies: $\neg \exists \sigma \in \Sigma^{\text{CHR}} : \sigma_f \rightarrowtail_{\mathcal{P}} \sigma$. In a failure state, the underlying solver $\mathcal{H}$ can prove $\mathcal{D}_{\mathcal{H}} \models \neg \bar{\exists}_{\emptyset} \mathbb{B}$ — such states are always final. A successful final state is a final state that is not a failure state, i.e. $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset} \mathbb{B}$. The set of final states is denoted by $\Sigma^{final} \subset \Sigma^{\text{CHR}}$.*

Definition 1.1.7 (finite derivation)**.** *Given a CHR program $\mathcal{P}$, a finite derivation d is a finite sequence $[\sigma_0, \sigma_1, \ldots, \sigma_n]$ of states where $\sigma_0 \in \Sigma^{init}$, $\sigma_n \in \Sigma^{final}$, and $\sigma_i \rightarrowtail_{\mathcal{P}} \sigma_{i+1}$*

for $0 \leq i < n$. If σ_n is a failure state, we say d has failed, otherwise d is a successful derivation.

Programs do not necessarily terminate, so derivations are not always finite:

Definition 1.1.8 (infinite derivation)**.** *An infinite derivation d_∞ is an infinite sequence $\sigma_0, \sigma_1, \ldots$ of states where $\sigma_0 \in \Sigma^{init}$ and $\sigma_i \rightarrowtail_{\mathcal{P}} \sigma_{i+1}$ for $i \in \mathbb{N}$.*

Two syntactically different execution states are essentially the same if they are renamings of another or if they are both failure states. We say they are variants, denoted by $\sigma \approx \sigma'$. For a more thorough discussion of variants and an axiomatic definition of it, the reader should consult Chapter 8.

We use $\#d$ to denote the length of a derivation: the length of a finite derivation is the number of transitions in the sequence; the length of an infinite derivation is ∞. A set of (finite or infinite) derivations is denoted by Δ. The set of all derivations in Δ that start with *initstate*$(\mathbb{G})$ is denoted by $\Delta|_{\mathbb{G}}$. We use $\Delta^{\mathcal{H}}_{\omega_t}(\mathcal{P})$ to denote the set of all derivations (in the ω_t semantics) for a given CHR program $\mathcal{P}$ and host language $\mathcal{H}$. We now define the relation $\rightsquigarrow_\Delta$: $\Sigma^{init} \rightarrow \Sigma^{\text{CHR}} \cup \{\infty\}$:

Definition 1.1.9 (Δ-output)**.** *State σ_n is a Δ-output of σ_0 if $[\sigma_0, \ldots, \sigma_n] \in \Delta$. We say σ_0 Δ-outputs σ_n and write $\sigma_0 \rightsquigarrow_\Delta \sigma_n$. If Δ contains an infinite derivation starting with σ_0, we say σ_0 has a non-terminating derivation. We denote this as follows: $\sigma_0 \rightsquigarrow_\Delta \infty$.*

Definition 1.1.10 (Δ-deterministic)**.** *The CHR program $\mathcal{P}$ is Δ-deterministic for input $I \subseteq \mathcal{G}^{\mathcal{H}}_{\mathcal{P}}$ if the restriction of $\rightsquigarrow_\Delta$ to initstate(I) is a function, modulo variant states (so for any input in I, all output states are variants of each other), and $\forall i \in I, d \in \Delta|_i$: if d is a successful derivation, then $\forall d' \in \Delta|_i : \#d = \#d'$.*

In other words, a program is Δ-deterministic if all derivations starting from a given input have the same result and all successful ones have the same length.

Example 1.1.11 (PRIMES is $\Delta^{\text{Prolog}}_{\omega_t}$-deterministic)**.** *Consider the CHR program* PRIMES *of Example 1.1.2 (page 19). This program is $\Delta^{Prolog}_{\omega_t}$-deterministic for input $\{\texttt{upto}(n) | n \in \mathbb{N}\}$. Although the order in which the transitions of ω_t are applied is not fixed for a given input, allowing different derivations, the derivation length and result is always the same.*

Of course not every CHR program is $\Delta^{\mathcal{H}}_{\omega_t}$-deterministic for its intended input:

Example 1.1.12 (LEQ is not $\Delta^{\text{Prolog}}_{\omega_t}$-deterministic)**.** *The CHR program* LEQ *of Example 1.1.1 (page 18) is not $\Delta^{Prolog}_{\omega_t}$-deterministic for all conjunctions of* `leq`*/2 constraints. Consider the input query "*`leq`*(A,B),* `leq`*(B,A)". One derivation consists of two* **Introduce** *steps followed by an* **Apply** *step using the antisymmetry rule, followed by a* **Solve** *step for "A = B". This derivation has length four. In another derivation, the transitivity rule is applied after the* **Introduce** *steps. This results in a longer derivation, or even an infinite derivation.*

1.1.7 Refined Operational Semantics ω_r

This subsection was added to provide a formal introduction to the refined operational semantics, as other chapters of the book refer to it. It is adapted from [Sneyers, 2008b, Chapter 4].

As a formal description of the standard compilation scheme (and thus also as a description of the actual behavior of most CHR systems), Duck et al. [2004] have introduced

1. **Solve.** $\langle [c|\mathbb{A}], \mathbb{S}' \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\omega_r,\mathcal{P}} \langle \mathbb{S} +\!\!+ \mathbb{A}, \mathbb{S}' \uplus \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$
   if c is a built-in constraint and $\mathbb{B}$ fixes the variables of $\mathbb{S}'$.

2. **Activate.** $\langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\omega_r,\mathcal{P}} \langle [c\#n\!:\!1|\mathbb{A}], \mathbb{S}', \mathbb{B}, \mathbb{T} \rangle_{n+1}$
   if c is a CHR constraint, where $\mathbb{S}' = \{c\#n\} \uplus \mathbb{S}$.

3. **Reactivate.** $\langle [c\#i|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\omega_r,\mathcal{P}} \langle [c\#i\!:\!1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$

4. **Drop.** $\langle [c\#i\!:\!j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\omega_r,\mathcal{P}} \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
   if there is no j-th occurrence of c in $\mathcal{P}$.

5. **Simplify.** $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
   $\rightarrowtail_{\omega_r,\mathcal{P}} \langle C +\!\!+ \mathbb{A}, H_1 \uplus \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
   if the j-th occurrence of the constraint c is d_j in a rule r in $\mathcal{P}$ of the form $r \mathbin{@} H'_1 \setminus H'_2, d_j, H'_3 \iff G \mid C$, a matching substitution θ exists such that $c = \theta(d_j)$, $chr(H_k) = \theta(H'_k)$ for $k = 1, 2, 3$, $\mathcal{D} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G)$, and $\mathbb{T} \not\ni h = (r, id(H_1), id(H_2 +\!\!+ c\#i +\!\!+ H_3))$.

6. **Propagate.** $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
   $\rightarrowtail_{\omega_r,\mathcal{P}} \langle C +\!\!+ [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \uplus H_1 \uplus H_2 \uplus \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
   if the j-th occurrence of the constraint c is d_j in a rule r in $\mathcal{P}$ of the form $r \mathbin{@} H'_1, d_j, H'_2 \setminus H'_3 \iff G \mid C$ a matching substitution θ exists such that $c = \theta(d_j)$, $chr(H_k) = \theta(H'_k)$ for $k = 1, 2, 3$, $\mathcal{D} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G)$, and $\mathbb{T} \not\ni h = (r, id(H_1 +\!\!+ c\#i +\!\!+ H_2), id(H_3))$.

7. **Default.** $\langle [c\#i\!:\!j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\omega_r,\mathcal{P}} \langle [c\#i\!:\!(j+1)|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
   if no other transition applies.

Figure 1.2: Transitions of the refined operational semantics ω_r

the refined operational semantics ω_r of CHR. It can be shown that ω_r is an instantiation of ω_t — and hence that the standard compilation scheme is correct w.r.t. the ω_t semantics, i.e. it applies only rules that may be applied and execution halts only if there are no more applicable rules.

The refined operational semantics uses a stack of constraints. When a new constraint arrives in the constraint store it is pushed on the stack. The constraint on top of the stack is called the *active* constraint. The active constraint searches for matching rules, in the order in which this constraint occurs in the program. The constraint is popped from the stack when all occurrences have been tried. When a rule fires, its body is executed immediately, from left to right, suspending the execution of the active constraint while the body is executed. When the constraint becomes topmost again, it resumes its search for matching clauses.

Definition 1.1.13 (occurrenced identified constraint)**.** *An occurrenced identified CHR constraint $c\#i\!:\!j$ is an identified constraint $c\#i$ annotated with an occurrence number j. This annotation indicates that only matches with occurrence j of constraint c are considered at this point in the execution.*

Definition 1.1.14 (ω_r execution state)**.** *An ω_r execution state σ is a tuple of the form $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$, where $\mathbb{S}$, $\mathbb{B}$, $\mathbb{T}$, and n represent the CHR store, the built-in store, the propagation history and the next free identity number just like before. semanticsz]$\mathbb{A}$execution stack*

(in an ω_r CHR execution state) The execution stack $\mathbb{A}$ is a sequence of constraints, identified CHR constraints and occurrenced identified CHR constraints, with a strict ordering where the top-most constraint is called active.

Execution in ω_r proceeds by exhaustively applying transitions from Figure 1.2 to the initial execution state until the built-in store is unsatisfiable or no transitions are applicable. Initial and final states are defined in the same way as in ω_t.

1.1.8 Other Operational Semantics

This subsection was added to give the reader a better overview of the available operational semantics of CHR. It is not from [Sneyers, 2008b], but from [Raiser, 2010].

The CHR community has formulated a plethora of operational semantics and extensions for CHR. The semantics discussed above are used throughout the remainder of this work and in this section we provide an (incomplete) overview of other operational semantics and extensions of CHR available in the literature.

Set-based Although CHR is primarily considered as a language for multiset rewriting, there is also an operational semantics ω_{set} available based on set rewriting. It is presented in [Sarna-Starosta and Ramakrishnan, 2007], which changes two other significant aspects of CHR as well: Firstly, tabled execution is considered in analogy to tabled Prolog variants. Secondly, the trivial non-termination problem is solved in a unique way different from the propagation history. The latter change is discussed in more detail in Chapter 8.

Disjunctive The committed-choice nature of CHR means that computations are made without backtracking, i.e. a rule application is never reversed or undone. This complicates formulating search algorithms in CHR. An extension which allows disjunctive bodies, and hence, backtracking over these alternatives, is called CHR$^\vee$[Abdennadher and Schütz, 1998, Abdennadher, 2001].

Probabilistic Another important extension of CHR allows probabilistic elements. PCHR [Frühwirth et al., 2002] is an early formulation that allows probabilistic selection of applicable rules. This idea has been taken further in the work on CHRiSM [Sneyers et al., 2009a, 2010a], which is a combination of CHR with the probabilistic Prolog dialect PRISM.

Concurrent CHR in its abstract form is well-suited for concurrent execution and prototype implementations exist which exploit this property (cf. [Frühwirth, 2009]). Lam and Sulzmann [2007] introduced a concurrent implementation in Haskell. It uses Haskell's support for shared transaction memory to resolve conflicting rule applications. The implementation was further refined in [Sulzmann and Lam, 2008] and is presented in more detail in Chapter 5.

CHR^2 A very recent proposal for a new operational semantics, called CHR^2, has been given by Van Weert [2010] in his thesis. It offers a combination of many features of existing extensions of CHR, like aggregates, rule priorities, or batch processing. However, there only exists a prototype implementation of CHR^2 and a glance at the inference rules given in [Van Weert, 2010] immediately reveals that it tremendously complicates formal analysis. As the goal of our thesis is to find an elegant formulation of the operational semantics, we will not discuss CHR^2 any further. However, the results given in Chapter 8 can be

extended analogously to existing CHR extensions, such as to come closer to an elegant formal representation of the features of CHR[2].

More details on available formulations and extensions can be found in [Frühwirth, 2009] and the surveys [Frühwirth, 1998, Sneyers et al., 2010b].

1.2 Program properties

For a more recent perspective on this topic, the original chapter was merged with a corresponding chapter from [Raiser, 2010] to provide an overview of the literature with regards to program analysis methods for different program properties.

Termination As CHR is Turing-complete [Sneyers et al., 2009b], the termination problem is undecidable in general. An early termination analysis method was derived from results for term rewriting systems: Given a mapping φ from states into $\mathbb{N}$, often called *measure function*, it suffices to prove that each rule application results in a state with a lower mapped number (cf. [Baader and Nipkow, 1998]). This technique was first adapted for termination analysis of CHR programs by Frühwirth [2000].

The results in [Frühwirth, 2000] apply only to CHR programs without propagation rules. A different approach, based on conditions imposed on the addition of constraints, has been introduced by Voets et al. [2007]. It specifically supports programs with propagation rules. The work of Pilozzi and De Schreye [2008] improves upon this: They link size-decreases of a different representation of CHR states to termination, which provides a strictly more powerful condition than both previous approaches.

Confluence If for a given CHR program, for all initial states, any ω_t derivation from that state results in the same final state, the program is called *confluent* [Abdennadher et al., 1999]. Confluence implies correctness, in the sense that the logical reading is consistent for confluent programs [Frühwirth, 1998].

Confluence is defined in terms of joinable execution states:

Definition 1.2.1 (joinable states)**.** *Execution states σ_1 and σ_2 are joinable if there exist states σ'_1 and σ'_2 such that $\sigma_1 \rightarrowtail^*_{\mathcal{P}} \sigma'_1$ and $\sigma_2 \rightarrowtail^*_{\mathcal{P}} \sigma'_2$ and $\sigma'_1 \approx \sigma'_2$.*

Definition 1.2.2 (confluence)**.** *A CHR program $\mathcal{P}$ is confluent if, for every initial state $\sigma \in \Sigma^{init}$, the following holds: if $\sigma \rightarrowtail^*_{\mathcal{P}} \sigma_1$ and $\sigma \rightarrowtail^*_{\mathcal{P}} \sigma_2$, then σ_1 and σ_2 are joinable.*

The above definition deviates slightly from the usual definitions in the literature: normally every execution state is considered instead of only the initial states. There are programs that are confluent according to the above definition but not according to the usual definition, where non-confluence originates from unreachable states. This issue has been addressed in a general way by introducing the notion of *observable confluence* [Duck et al., 2007].

There is a decidable, sufficient and necessary test for confluence (according to the usual definition) of terminating programs [Frühwirth, 1998]. Recently, the topic of confluence received renewed attention because certain problems and limitations of this confluence test have surfaced. Firstly, the test is only applicable to terminating programs. Raiser and Tacchella [2007] investigated confluence of non-terminating

programs. Secondly, many programs that are in practice confluent fail the test because non-confluence originates from unreachable states. The framework of observable confluence [Duck et al., 2007] allows a restriction to reachable states. It was further refined in [Raiser, 2010], based on the formalisms developed in Chapter 8. This refinement also serves as the foundation for an implementation of a confluence checker given in [Langbein et al., 2010].

Completion Completion is a method for modifying a non-confluent program such that it becomes confluent. Early results for term rewriting systems date back to the work from Knuth and Bendix [1970]. Abdennadher and Frühwirth [1998] showed how to do *completion* of CHR programs. Completion is a technique to transform a non-confluent program into a confluent one by adding rules. It allows extension, modification and specialization of existing programs.

Modularity Multiple CHR programs can be considered as modules to be combined in two different ways. A *flat union* simply merges all rules into a single CHR program. This kind of modularity has been investigated by Abdennadher and Frühwirth [2004] and it has been shown, that in general confluence and termination of individual programs is not preserved for their union.

The second possibility is a *hierarchical* approach, which allows CHR constraints from one program to be reused in other programs as built-in constraints. This requires that the implication of such a constraint can be checked. Schrijvers et al. [2006a] investigated checking them automatically, whereas Fages et al. [2008] proposed user-definable rules for implication checking via ask and tell constraints.

Complexity Frühwirth [2002b] was the first to investigate automatic time complexity analysis for CHR programs in [Frühwirth, 2002b] and [Frühwirth, 2002a]. However, this analysis is based on a naive CHR compiler without support for important optimizations, hence, the approach yields only weak upper boundaries.

As CHR rule heads can contain multiple constraints, it is incessant to optimize the code required for matching these head constraints to constraints in the store. A naive approach, which would try to match each head constraint with each constraint in the store, leads to exorbitantly slow runtimes. Therefore, CHR research produced a significant body of work on optimizations of CHR execution. A summary of the techniques developed up until 2005 is available in [Schrijvers, 2005]. Complexity analysis is discussed in more detail in Chapter 7.

Computational Power A notable line of research was the investigation of the computational power of CHR, which culminated in the PhD thesis from Sneyers [2008b]. This work assumes a CHR compiler that supports several important optimization techniques, which leads to the following significant result: any algorithm can be implemented in CHR in optimal time and space complexity.

After the proof that CHR in its general form is Turing-complete [Sneyers et al., 2005], restrictions of CHR, for example, to single-headed rules or limited built-in theories, have been investigated. This lead to the discovery of several Turing-complete subclasses of CHR [Sneyers, 2008a, Sneyers et al., 2009b, Gabbrielli et al., 2010, Mauro et al., 2010]. The original results from [Sneyers, 2008b] are presented in more detail in Chapter 6.

1.3 CHR systems

Since the conception of CHR, a large number of CHR systems (compilers, interpreters and ports) have been developed. In particular, in the last ten years the number of systems has exploded. Figure 1.3 presents a timeline of system development, branches and influences. In this section we briefly discuss these systems, grouped by host language paradigm.

1.3.1 CHR(LP)

Logic Programming is a natural (or at least traditional) host language paradigm for CHR. Hence, it is not surprising that the CHR(Prolog) implementations are the most established ones. Holzbaur and Frühwirth [2000b] have laid the groundwork with their general compilation scheme for Prolog. This compilation scheme (cf. Chapter 2) was first implemented in SICStus Prolog by Holzbaur, and later further refined in HAL by Holzbaur et al. [2005] and in hProlog by Schrijvers and Demoen [2004]. Another system directly based on the work of Holzbaur and Schrijvers is the CHR library for SiLCC by Bouissou [2004]. SiLCC is a programming language based on linear logic and concurrent constraint programming. All of these systems compile CHR programs to host language programs. The only available *interpreter* for CHR(Prolog) is TOYCHR[1].

1.3.2 CHR(FP)

As type checking is one of the most successful applications of CHR in the context of Functional Programming (see section 1.7.3), several CHR implementations were developed specifically for this purpose. Most notable is the Chameleon system [Stuckey and Sulzmann, 2005] which features CHR as the programming language for its extensible type system. Internally, Chameleon uses the HaskellCHR implementation[2]. The earlier HCHR prototype [Chin et al., 2003] had a rather heavy-weight and impractical approach to logical variables.

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

With the advent of software transactional memories (STM) in Haskell, two prototype systems with parallel execution strategies have been developed: STMCHR[3] and Concurrent CHR [Lam and Sulzmann, 2007]. These systems are currently the only CHR implementations that exploit the inherent parallelism in CHR programs.

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

1.3.3 CHR(Java) and CHR(C)

Finally, CHR systems are also available for both Java and C. For a detailed discussion on the different conceptual and technical challenges encountered when embedding CHR into an imperative host language, see [Van Weert et al., 2008].

[1] by Gregory J. Duck, 2003. Download: http://www.cs.mu.oz.au/~gjd/toychr/
[2] by Gregory J. Duck, 2004. Download: http://www.cs.mu.oz.au/~gjd/haskellchr/
[3] by Michael Stahl, 2007. Download: http://www.cs.kuleuven.be/~dtai/projects/CHR/

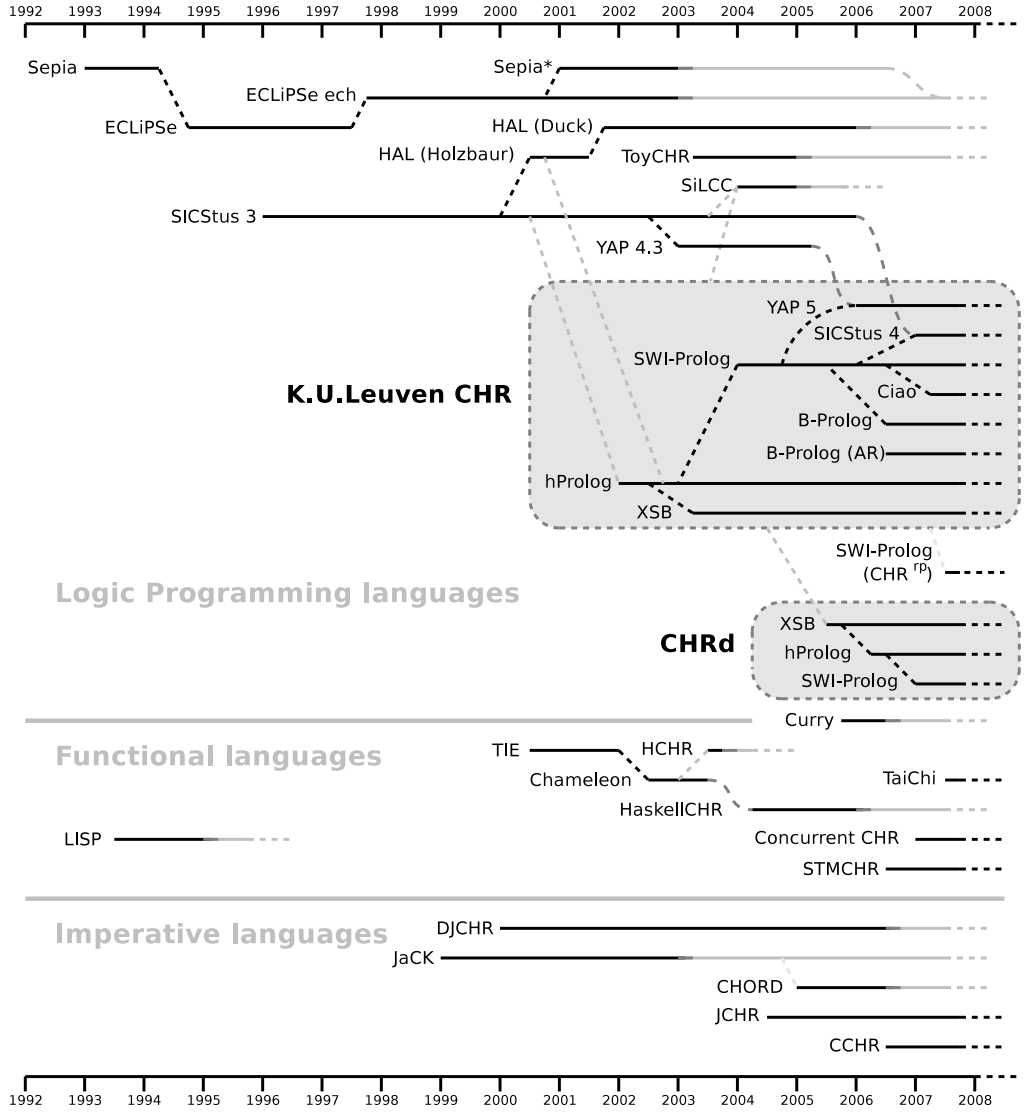


Figure 1.3: Timeline of CHR systems

CHR(Java). The earliest CHR(Java) system is the Java Constraint Kit (JaCK) by Abdennadher [2001] and others [Abdennadher et al., 2002, Schmauß, 1999]. DJCHR (Dynamic JCHR; [Wolf, 2001a]) is an implementation of adaptive CHR (see Section 1.6.3). The incremental adaptation algorithm underlying DJCHR maintains *justifications* for rule applications and constraint additions. The Leuven JCHR system[4] [Van Weert et al., 2005] focusses on performance and integration with the host language. It is currently one of the most efficient CHR systems available. A fourth CHR(Java) system is called CHORD (Constraint Handling Object-oriented Rules with Disjunctive bodies)[5].

CHR(C). CCHR [Wuille et al., 2007] implements CHR for C. It is an extremely efficient CHR system and uses a syntax that is intuitive to both CHR adepts and imperative programmers.

1.3.4 The Leuven CHR system

The Leuven CHR system[6] is a state-of-the-art CHR system, developed by Tom Schrijvers and others at the K.U.Leuven. It is available in seven different Prolog systems: hProlog[7] [Schrijvers and Demoen, 2004], SWI-Prolog[8] [Schrijvers et al., 2005], XSB[9] [Schrijvers et al., 2003, Schrijvers and Warren, 2004], YAP[10], B-Prolog[11] [Schrijvers et al., 2006b], SICStus Prolog[12], and Ciao Prolog[13].

One of the distinguishing features of the Leuven CHR system is the option to add type and mode declarations of constraint arguments. We discuss these declarations in Section 1.4.1 and they will be used throughout the text.

1.4 Example CHR programs

In this section we present some examples of CHR programs. Some of these examples will be reconsidered in later chapters, for instance for benchmarking purposes.

1.4.1 Programming in CHR, in practice

Before we discuss the examples, we first explain how to try these examples in practice, in the Leuven CHR system. Every CHR(Prolog) program starts by loading the CHR library:

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

In program listings we do not explicitly include this line. Next, we have to declare which predicates are CHR constraints. The keyword `chr_constraint` is used for these declarations. For example, if the only CHR constraints used in a program are called `foo`, with two arguments, and `bar`, with three arguments, we need the following declaration:

[4] Leuven JCHR system home page: http://dtai.cs.kuleuven.be/projects/CHR/JCHR/
[5] by Jairson Vitorino and Marcos Aurelio, 2005, `http://chord.sourceforge.net/`
[6] Leuven CHR system home page: http://www.cs.kuleuven.be/~toms/Research/CHR/
[7] by Bart Demoen, http://www.cs.kuleuven.be/~bmd/hProlog/
[8] by Jan Wielemaker, http://www.swi-prolog.org/
[9] by David S. Warren et al, http://xsb.sf.net/
[10] by Vítor Santos Costa et al, http://www.ncc.up.pt/~vsc/Yap/
[11] by Neng-Fa Zhou, http://www.probp.com
[12] by Mats Carlsson et al, http://www.sics.se/isl/sicstuswww/site/
[13] by Manuel Hermenegildo et al, http://www.ciaohome.org/

Listing 1.3: SUM: Sum of the elements of a list

```
:- chr_type list(T) ---> [] ; [T | list(T)].
:- chr_constraint sum(+list(int), ?int).

sum([],S) <=> S = 0.
sum([X|Xs],S) <=> sum(Xs,S2), S is X + S2.
```

```
:- chr_constraint foo/2, bar/3.
```

If a rule head contains an undeclared constraint, the Leuven CHR system produces an error message. In program listings we sometimes omit the constraint declarations, since they can easily be reconstructed by looking at the rule heads.

Mode declarations

The Leuven CHR system also supports a more advanced way to declare constraints. Instead of only giving the arity of each constraint predicate, the programmer can also provide the *mode* of each constraint argument. The default mode is "?", which corresponds to arbitrary argument instantiation. If an argument has mode "+", it should correspond to a ground term. The mode "-" indicates that the argument is an unbound variable. A typical example is the following:

```
:- chr_constraint address(+,+), find(+,?).
address(Person,Address) \ find(Person,Where) <=> Where = Address.
```

Mode declarations are crucial in several compiler optimizations; often performance can significantly be boosted by adding accurate mode declarations.

Type declarations

Even more information about the constraint arguments can optionally be declared by means of type declarations. The default type is `any`; other built-in types are `int`, `float`, `number`, and `natural`. User-defined types can be declared using the keyword `chr_type`. Algebraic data type in the style of Haskell or Mercury are supported. The syntax for type definitions is "`type ---> body`", where the right hand side is a disjunction of the different constructors. Type aliases can be defined using "`==`". We give some examples of type definitions:

Example 1.4.1 (type definitions)**.**

```
:- chr_type color ---> red ; green; blue.
:- chr_type cnode ---> node - color.
:- chr_type node == int.
:- chr_type edge ---> e(node,node).
:- chr_type tree ---> leaf(node) ; branch(node, tree, tree).
:- chr_type tree(T) ---> leaf(T) ; branch(T, tree(T), tree(T)).
```

The Leuven CHR system performs both static and dynamic type-checking. Type information is also used in some compiler optimizations. As a simple example, consider the program SUM of Listing 1.3. Erroneous queries like `sum([4,7,x],S)` are detected by dynamic type checks, and an error message is given:

Listing 1.4: FIBONACCI: Top-down computation of Fibonacci numbers

```
:- chr_constraint fib(+,?).

fib(N,C)#passive \ fib(N,V) <=> var(V) | V=C.

fib(0,M) ==> M=0.
fib(1,M) ==> M=1.
fib(N,M) ==> N>1 | fib(N-1,A), fib(N-2,B), M is A+B.
```

```
ERROR: Type error: 'int' expected, found 'x' (CHR Runtime Type Error)
```

1.4.2 Fibonacci numbers

The Fibonacci numbers are named after the medieval Italian mathematician Leonardo of Pisa, also known as Fibonacci. The sequence is recursively defined as follows:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1 \end{cases}$$

In other words, every Fibonacci number is the sum of the two previous Fibonacci numbers. The sequence starts as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Example 1.4.2 (Fibonacci numbers)**.** *The program* FIBONACCI*, shown in Listing 1.4, can be used to compute Fibonacci numbers. Starting with a query of the form* `fib(`n`,Q)`*, all Fibonacci numbers up to the n-th are computed and* `Q` *gets the value fib*(n)*.*

1.5 The union-find algorithm

A disjoint-set data structure can be used to efficiently maintain an equivalence relation — for example, variable unifications in Prolog. An equivalence relation on some set corresponds to a partition of that set into equivalence classes. A disjoint-set data structure should provide (at least) the following three operations:

- `make(X)` : add a new element `X` (in a new singleton equivalence class);
- `union(X,Y)` : assert that elements `X` and `Y` are equivalent. Their equivalence classes, if different, should be merged;
- `find(X,R)` : find a representative `R` of the equivalence class of `X`. All members of an equivalence class should have the same representative, so this operation can be used to determine whether two elements are equivalent.

The union-find algorithm of Tarjan and van Leeuwen [1984] can be shown to implement this data structure with optimal time complexity. Essentially, the disjoints sets are represented as trees, where the root node is the representative. The `union`/2 operation adds an edge between the root of one tree and the root of the other tree. The `find`/2 operation follows the edges until the root is found.

A naive implementation of the union-find algorithm is given in Listing 1.5. The naive version of the union-find algorithm still takes logarithmic time per operation on average.

Listing 1.5: N-UNION-FIND: Naive implementation of the union-find algorithm

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

f1 @ edge(A,B) \ find(A,X) <=> find(B,X).
f2 @ root(B) \ find(B,X) <=> X=B.

l1 @ link(A,A) <=> true.
l2 @ root(B) \ link(A,B), root(A) <=> edge(A,B).
```

Listing 1.6: UNION-FIND: Optimal implementation of the union-find algorithm

```
:- chr_constraint union(+,+), find(+,?), make(+), link(+,+),
                  edge(+,+), root(+,+).
make(A) <=> root(A,0).
union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

% path compression with immediate update
% (thanks to logical variable)
f1 @ find(A,X), edge(A,B) <=> find(B,X), edge(A,X).
f2 @ root(B,_) \ find(B,X) <=> X=B.

% union-by-rank
l1 @ link(A,A) <=> true.
l2 @ link(A,B), root(A,NA), root(B,NB) <=> NA>=NB |
            edge(B,A), NA1 is max(NA,NB+1), root(A,NA1).
l3 @ link(B,A), root(A,NA), root(B,NB) <=> NA>=NB |
            edge(B,A), NA1 is max(NA,NB+1), root(A,NA1).
```

Two techniques can be used to improve the performance of the algorithm; if combined, the average time per operation can be reduced to $O(\alpha(n))$, the inverse Ackermann function of n, which can be considered constant for all practical purposes. The first technique is called *union-by-rank* and consists of maintaining the size of each tree and modifying the `union`/2 operation such that it always links the smaller tree to the bigger tree. The second technique is called *path compression*: after every `find`/2 operation, all edges on the path to the root are modified so that every node on the path is linked directly to the root. Listing 1.6 gives an optimal implementation of the union-find algorithm. Both programs are due to Schrijvers and Frühwirth [2006].

1.5.1 Sudoku puzzle solver

Sudoku is a Japanese puzzle which relatively recently attained international popularity. Sudoku is played on a 9x9 board, consisting of 9 smaller 3x3 boxes. Initially, most board cells are empty and some have a value between 1 and 9. The aim is to fill the empty cells such that the same number does not occur more than once on every row, column, and box. Normally there is only one valid solution.

Listing 1.7: SUDOKU: Solver for Sudoku puzzles

```
:- chr_constraint possible(+,+,+,+,+,+), fill(+),
                  fixed(+,+,+,+,+).

search_ff    @ fill(N), possible(A,B,C,D,N,L)#passive
              <=> member(V,L), fixed(A,B,C,D,V), fill(1).
next_ff      @ fill(N) <=> fill(N+1).

same_column @ fixed(_,B,_,D,V) \ possible(A,B,C,D,N,L)#passive
              <=> select(V,L,L2) | N>1, possible(A,B,C,D,N-1,L2).
same_row     @ fixed(A,_,C,_,V) \ possible(A,B,C,D,N,L)#passive
              <=> select(V,L,L2) | N>1, possible(A,B,C,D,N-1,L2).
same_box     @ fixed(A,B,_,_,V) \ possible(A,B,C,D,N,L)#passive
              <=> select(V,L,L2) | N>1, possible(A,B,C,D,N-1,L2).

solve :- fill(1).
```

Example 1.5.1 (Sudoku)**.** *Listing 1.7 shows an implementation of a Sudoku puzzle solver based on Thom Frühwirth's rewrite of Jon Murua González' and Henning Christiansen's program.*

In the SUDOKU program, every board cell is identified using 4 coordinates: the first two refer to one 3x3 box, the other two give the position inside that box. The first four arguments of the constraints `fixed`/5 and `possible`/6 refer to the coordinates of a board cell. The constraint `fixed`/5 indicates that the value of a board cell is known; in a query, this constraint is used to input the given cells. If the value of a board cell is not yet known, the constraint `possible`/6 contains the number of remaining possible values and the list of values. For every empty board cell, the query should contain a constraint of the form `possible(...,9,[1,2,3,4,5,6,7,8,9])`.

The rules `same_column`, `same_row`, and `same_box` are very similar: if a board cell is known, its value is removed from the list of possibilities of any board cell in the same column (row, box) via the Prolog built-in `select/3`; if this was the last possibility (so `N=1`) then the program fails. These three rules remove all possible values that are directly in violation with the rules of Sudoku. The first two rules implement a search using the *first-fail* principle, that is, unknown board cell with few remaining possible values are tried first. The search starts with the `fill(1)` constraint. If there is a cell which has only one possible value, then the `possible`/6 constraint is replaced with a `fixed`/5 constraint, and then another cell is tried with only one possible value. Note that the new `fixed`/5 constraints may cause further applications of the `same_*` rules. If no more cells with only one possible value are found, then the rule `next_ff` kicks in, adding `fill(2)` which looks for a cell with only two possible values, and so on. If there is more than one possible value, the call to `member`/2 in the first rule nondeterministically chooses one of the values. If the chosen value turns out to be wrong (i.e. it leads to failure later on), Prolog will backtrack and try a different choice.

1.6 Extensions of CHR

Over the years, some additional language features have been proposed for CHR. In this section we give a brief overview of some of them.

1.6.1 Disjunction and search

Most constraint solvers require search next to constraint simplification and propagation. However, pure CHR does not offer any support for search. Abdennadher and Schütz [1998] propose a solution to this problem: an extension of CHR with disjunctions in rule bodies (see also Abdennadher [2000, 2001]). The resulting language is denoted $\text{CHR}^{\vee}$ (pronounced "CHR-or"). Any (pure) Prolog program can be rephrased as an equivalent $\text{CHR}^{\vee}$ program (Abdennadher [2000, 2001]). Implementation-wise, disjunction comes for free in CHR(Prolog) by means of the built-in Prolog disjunction and search mechanism.

In CHR(Java) systems, unlike in CHR(LP) systems, the host language does not provide search capabilities. The flexible specification of intelligent search strategies has therefore received considerable attention in several CHR(Java) systems [Krämer, 2001, Wolf, 2005]. In these systems, the search strategies are implemented and specified in the host language itself, orthogonally to the actual CHR program.

1.6.2 Negation and aggregates

CHR programmers often want to test for the absence of constraints. CHR was therefore extended with *negation as absence* by Van Weert et al. [2006]. Negation as absence was later generalized to a much more powerful language feature, called *aggregates* [Sneyers et al., 2007]. Aggregates accumulate information over unbounded portions of the constraint store. Predefined aggregates include `sum`, `count`, `findall`, and `min`. The proposed extension also features nested aggregate expressions over guarded conjunctions of constraints, and application-tailored user-defined aggregates. Aggregates lead to increased expressivity and more concise programs. An implementation based on source-to-source transformations [Van Weert et al., 2008] is available. The implementation uses efficient incremental aggregate computation, and empirical results show that the desired runtime complexity is attainable with an acceptable constant time overhead.

As an example of nested aggregate expressions, consider the following rule:

```
eulerian, forall(node(N),(
              count(edge(N,_),X), count(edge(_,N),X)
          )) <=> writeln(graph_is_eulerian).
```

The above rule is applicable if for every constraint `node(N)`, the number of outgoing edges in `N` equals the number of incoming edges (i.e. if the first number is `X`, the other number must also be `X`).

1.6.3 Adaptive CHR

Constraint solving in a continuously changing, dynamic environment often requires immediate adaptation of the solutions, i.e. when constraints are added or removed. By nature, CHR solvers already support efficient adaptation when constraints are added. Wolf [1999], Wolf et al. [2000] introduces an extended incremental adaptation algorithm which is capable of adapting CHR derivations after constraint deletions as well. This algorithm is further improved by Wolf [2000a] with the elimination of local variables using *early projection*. An efficient implementation exists in Java (Wolf [2001a,b]; cf. Section 1.3.3).

1.6.4 Solver hierarchies

While the theory of the CHR language generally considers arbitrary built-in solvers, traditional CHR implementations restrict themselves to the Herbrand equality constraint solver, with very little, if any, support for other constraint solvers.

Duck et al. [2003] show how to build CHR solvers on top of arbitrary built-in constraint solvers by means of *ask* constraints. The ask constraints signal the CHR solver when something has changed in the built-in store with respect to variables of interest. Then the relevant CHR constraints may be reactivated.

Schrijvers et al. [2006a] provide an automated means for deriving ask versions of CHR constraints. In the approach of Fages et al. [2008], which is called CHRat, the programmer is supposed to (manually) implement both ask and tell versions of each constraint. Both approaches are aimed at adding a form of modularity to CHR. In this way full hierarchies of constraint solvers can be written, where one solver serves as the built-in solver for another solver.

1.7 Applications of CHR

In this section, we give a very brief overview of recent applications of CHR. A more exhaustive overview can be found in [Sneyers et al., 2010b].

1.7.1 Constraint solvers

CHR was originally designed specifically for writing constraint solvers. Some recent examples are the following:

Lexicographic order. Frühwirth [2006b] presented a constraint solver for a lexicographic order constraint in terms of inequality constraints offered by the underlying solver.

Rational trees. Meister et al. [2006] presented a solver for existentially quantified conjunctions of non-flat equations over rational trees. Djelloul et al. [2007b] use this solver for rational trees as a component of a more general solver for (quantified) first-order constraints over finite and infinite trees.

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

Solvers derived from union-find. Frühwirth [2006a] has proposed linear-time algorithms for solving certain boolean equations and linear polynomial equations in two variables. These solvers are derived from the classic union-find algorithm [Schrijvers and Frühwirth, 2006].

Soft constraints. An important class of constraints are the so-called *soft constraints* which are used to represent preferences amongst solutions to a problem. Unlike hard (required) constraints which must hold in any solution, soft (preferential) constraints must only be satisfied as far as possible. Bistarelli et al. [2004] have presented a series of constraint solvers for (mostly) extensionally defined finite domain soft constraints.

Another well-known formalism for describing over-constrained systems is that of *constraint hierarchies*, where constraints with hierarchical strengths or preferences can be specified, and non-trivial error functions can be used to determine the solutions. Wolf [2000b] has proposed an approach for solving dynamically changing constraint hierarchies.

Scheduling. Abdennadher and Marte [2000] have successfully used CHR for scheduling courses at the university of Munich. Their approach is based on soft constraints to deal with teacher's preferences. The related problem of assigning classrooms to courses, given a timetable, is dealt with in [Abdennadher et al., 2000].

Spatio-temporal reasoning. In the context of autonomous mobile robot navigation, a crucial research topic is automated qualitative reasoning about spatio-temporal information, including orientation, named or compared distances, cardinal directions, topology and time. The use of CHR for spatio-temporal reasoning has received considerable research attention. We mention in particular the contributions of Escrig et al. (Escrig and Toledo [1998a,b]; [Cabedo and Escrig, 2003]).

Multi-agent systems. FLUX (Thielscher [2002, 2005]) is a high-level programming system, implemented in CHR and based on fluent calculus, for cognitive agents that reason logically about actions in the context of incomplete information. An interesting application of this system is FLUXPLAYER [Schiffel and Thielscher, 2007], which won the 2006 General Game Playing (GGP) competition at AAAI'06. Seitz et al. [2002] and Alberti et al. (Alberti and Daolio et al. 2004; Alberti and Gavanelli et al. 2004, 2006) have also applied CHR in the context of multi-agent systems. Lam and Sulzmann [2006] have explored the use of CHR as an agent specification language, founded on CHR's linear logic semantics (see Section 1.1.4).

Data integration. One of the core problems related to the so-called *Semantic Web* is the integration and combination of data from diverse information sources. Bressan and Goh [1998] have described an implementation of the COIN (COntext INterchange) mediator that uses CHR for solving integrity constraints. In more recent work, CHR has been used for implementing an extension of the COIN framework, capable of handling more data source heterogeneity [Firat, 2003]. Badea et al. [2004] have presented an improved mediator-based integration system.

Description logic. The Web Ontology Language (OWL) is based on Description Logic (DL). Various rule-based formalisms have been considered for combination and integration with OWL or other description logics. Frühwirth [2007] has proposed a CHR-based approach to DL and DL rules.

1.7.2 Automatic solver generation

Many authors have investigated the automatic generation of CHR rules from a formal specification. Most works consider extensionally defined constraints over (small) finite domains as the specification.

Apt and Monfroy [2001] have shown how to derive propagation rules from an extensional definition of a finite domain constraint. As an extension, Brand and Monfroy [2003] have proposed to transform the derived rules to obtain stronger propagation rules. Brand [2002] has proposed a method to eliminate redundant propagation rules.

Abdennadher and Rigotti [2004] have also derived propagation rules from extensionally defined constraints. In contrast to the approach of Apt and Monfroy [2001], rules are

assembled from given parts and propagation rules are transformed into simplification rules if possible. In [Abdennadher and Rigotti, 2005] the approach is extended to intensional constraint definitions, where constraints are defined by logic programs. The latter is further extended in [Abdennadher and Sobhi, 2008] to symbolically derive rules from the logic programs, rather than from given parts.

1.7.3 Type systems

CHR's aptness for symbolic constraint solving has led to many applications in the context of type system design, type checking and type inference. While the basic Hindley-Milner type system requires no more than a simple Herbrand equality constraint, more advanced type systems require custom constraint solvers.

The most successful use of CHR in this area is for Haskell type classes. Type classes are a principled approach to ad hoc function overloading based on type-level constraints. By defining these type class constraints in terms of a CHR program [Stuckey and Sulzmann, 2005] the essential properties of the type checker — soundness, completeness and termination — can be established. Moreover, various extensions, such as multi-parameter type classes [Sulzmann et al., 2006] and functional dependencies [Sulzmann et al., 2007] are easily expressed.

1.7.4 Abduction

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

The earliest work connecting CHR with abduction is that of Abdennadher and Christiansen [2000]. It shows how to model logic programs with abducibles and integrity constraints in CHR$^\vee$. The HYPROLOG system of Christiansen and Dahl [2005] combines abductive reasoning and abductive-based logic programming in one system. Christiansen [2006] has also proposed the use of CHR for the implementation of *global abduction*, an extended form of logical abduction for reasoning about a dynamic world.

1.7.5 Computational linguistics

CHR allows flexible combinations of top-down and bottom-up computation [Abdennadher and Schütz, 1998], and abduction fits naturally in CHR as well (see Section 1.7.4). It is therefore not surprising that CHR has proven a powerful implementation and specification tool for language processors.

The most successful approach to CHR-based language processing is that of *CHR grammars* (CHRG), a highly expressive, bottom-up grammar specification language proposed by Christiansen [2005]. Christiansen recognizes that the CHR language itself can be used as a powerful grammar formalism. CHRG's, built as a relatively transparent layer of syntactic sugar over CHR, are to CHR what DCG's are to Prolog.

Applications of CHRG. Using CHRG, Dahl and Blache [2005] have developed directly executable specifications of property grammars. In [Dahl and Gu, 2006], an extension of this approach is used to extract concepts and relations from biomedical texts. Dahl and Voll [2004] have generalized the property grammar parsing methodology into a general concept formation system. Applications of this formalism include early lung cancer diagnosis [Barranco-Mendoza, 2005, Chapter 4], error detection and correction of radiology

reports obtained from speech recognition [Voll, 2006, Section 5.2.8], and the analysis of biological sequences [Bavarian and Dahl, 2006].

1.7.6 Testing and verification

Another application domain for which CHR has proved useful is software testing and verification. Ribeiro et al. [2000] have presented a CHR-based tool for detecting security policy inconsistencies. Lötzbeyer and Pretschner [2000], Pretschner et al. [2004] have proposed a model-based testing methodology, in which test cases are automatically generated from abstract models using CLP and CHR. They considered the ability to formulate arbitrary test case specifications by means of CHR to be one of the strengths of their approach. Gouraud and Gotlieb [2006] have used a similar approach for the automatic generation of test cases for the Java Card Virtual Machine (JCVM). A formal model of the JCVM is automatically translated into CHR, and the generated CHR program is used to generate test cases.

More of an exploration than testing application is the JmmSolve framework presented in [Schrijvers, 2004]. Its purpose is to explore and test the behavior of declarative memory models for Java, based on the Concurrent Constraint-based Memory Machines proposal of Vijay Saraswat.

1.8 Related formalisms

For a more recent overview of related formalisms, the original version of this section was merged with a corresponding section from [Raiser, 2010].

The proposal of CHR as a lingua franca for rewriting systems arrived after numerous comparisons between CHR and other formalisms have been undertaken. In this section, we recapitulate selected results from this line of research.

Set-based Formalisms

A comprehensive comparison of RETE-based systems with CHR, focusing on efficient execution, was given by Van Weert [2009]. Business rules have influenced the early CHR compiler implementations, but the current systems are based on numerous research results for improving efficiency. A summary of the techniques applied in CHR compilation is given by [Schrijvers, 2005] in his PhD thesis. Together with more recent optimizations, Van Weert [2009] has shown that execution of CHR is faster than traditional RETE-based systems by several magnitudes.

De Koninck [2009] investigated the close relation between CHR and logical algorithms (LA). LA is a formalism proposed by Ganzinger and McAllester [2002] in order to alleviate the problem of determining runtime complexity of logic programs. De Koninck [2009] successfully embedded LA into CHR with rule priorities, which allowed him to transfer a meta-complexity result, available for LA, to a subset of CHR with rule priorities. Additionally, he provided a mapping from CHR with rule priorities into regular CHR, which in turn made his embedding the first actual implementation of LA.

Logical Formalisms

Apart from the already discussed first-order and linear-logic declarative semantics, other logical formalisms have been considered. This line of research was pursued mainly by Meister [2008].

He implemented a fragment of frame-logic, which is an object-oriented extension of classical first-order logic, in CHR. Furthermore, he gave a transaction logic semantics for CHR.

Term Rewriting

CHR is closely related to term rewriting. The main difference is that CHR rewrites a flat set of constraints, whereas in term rewriting nested terms can be rewritten. It is possible to simulate term rewriting in CHR by a simple flattening function, as explained in [Frühwirth, 2009]. However, this only works for linear term rewriting systems.

Raiser and Frühwirth [2008] instead considered term graph rewriting and provided the necessary folding rules in CHR for sharing term structures. This work is based on prior research by Plump [1993] on jungle evaluation, and a general treatment of term graph rewriting is available in [Ohlebusch, 2002].

Term rewriting is the basis of functional programming and CHR has also been applied to the problem of type checking and inference [Alves and Florido, 2002, Stuckey et al., 2006]. Finally, Martinez [2010] compared linear concurrent constraint programming with CHR, which resulted in an encoding of the λ-calculus in CHR.

Graph-based Formalisms

There only exist few research results in this direction prior to [Raiser, 2010], which compares graph transformation system and CHR.

The comparison with term graph rewriting was already mentioned above. Another interesting work is given by Betz [2007], who compared colored Petri nets to CHR. A subset of these can be translated into CHR, and there also exists a sound and complete encoding of place/transition nets.

Join-Calculus

The join-calculus is a calculus for concurrent programming, with both stand-alone implementations and extensions of general purpose languages, such as JoCaml (OCaml), Join Java and Polyphonic C#.

Sulzmann and Lam [2007] propose a Haskell language extension for supporting join-calculus-style concurrent programming, based on CHR. Join-calculus rules, called chords, are essentially guardless simplification rules with linear match patterns. In a linear pattern, different head conjuncts are not allowed to share variables. Hence, CHR offers considerably increased expressivity over the join-calculus: propagation rules, general guards and non-linear patterns.

Other Formalisms

We explained above that we will not elaborate on all formalisms that have been compared to CHR. For more complete surveys, the reader should consult [Frühwirth, 2009] and [Sneyers et al., 2010b]. In addition to the above, these discuss comparisons of CHR with ACD term rewriting, equivalent transformation rules, production rules, event-condition-action rules, GAMMA, functional programming, deductive databases, Prolog, (concurrent) constraint logic programming, and more.

Bibliography

Slim Abdennadher. A language for experimenting with declarative paradigms. In T. Frühwirth et al., editors, *RCoRP'00(bis): Proc. 2nd Workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, September 2000.

Slim Abdennadher. Rule-based constraint programming: Theory and practice. Habilitationsschrift, July 2001. Inst. of Comp. Sc., LMU, Munich, Germany.

Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *FQAS'00: Proc. 4th Intl. Conf. Flexible Query Answering Systems*, pages 141–152, Warsaw, Poland, October 2000.

Slim Abdennadher and Thom Frühwirth. On completion of Constraint Handling Rules. In M. J. Maher and J.-F. Puget, editors, *CP'98*, volume 1520 of *LNCS*, pages 25–39, Pisa, Italy, October 1998. Springer. ISBN 3-540-65224-8.

Slim Abdennadher and Thom Frühwirth. Integration and optimization of rule-based constraint solvers. In M. Bruynooghe, editor, *LOPSTR'03*, volume 3018 of *LNCS*, pages 198–213, Uppsala, Sweden, 2004. Springer. URL `http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/Papers/paper3.pdf`.

Slim Abdennadher and Michael Marte. University course timetabling using Constraint Handling Rules. In *J. Applied Artificial Intelligence, Special Issue on Constraint Handling Rules* Holzbaur and Frühwirth [2000a], pages 311–325.

Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based constraint solvers over finite domains. *ACM TOCL*, 5(2):177–205, 2004. ISSN 1529-3785.

Slim Abdennadher and Christophe Rigotti. Automatic generation of CHR constraint solvers. In *TPLP* Abdennadher et al. [2005], pages 403–418.

Slim Abdennadher and Heribert Schütz. CHR$^\vee$, a flexible query language. In Andreasen et al. [1998], pages 1–14.

Slim Abdennadher and Ingi Sobhi. Generation of rule-based constraint solvers: Combined approach. In King [2008].

Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999. ISSN 1383-7133. doi: http://dx.doi.org/10.1023/A:1009842826135.

Slim Abdennadher, Matthias Saft, and Sebastian Will. Classroom assignment using constraint logic programming. In *PACLP'00: Proc. 2nd Intl. Conf. and Exhibition on Practical Application of Constraint Technologies and Logic Programming*, Manchester, UK, April 2000.

Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauß. JACK: A Java Constraint Kit. In Hanus [2002], pages 1–17.

Slim Abdennadher, Thom Frühwirth, and Christian Holzbaur, editors. *Special Issue on Constraint Handling Rules*, volume 5(4–5) of *TPLP*, July 2005.

Marco Alberti, Davide Daolio, Paolo Torroni, Marco Gavanelli, Evelina Lamma, and Paola Mello. Specification and verification of agent interaction protocols in a logic-based system. In H. Haddad et al., editors, *SAC'04: Proc. 19th ACM Symp. Applied Computing*, pages 72–78, Nicosia, Cyprus, March 2004a. ACM Press.

Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Specification and verification of agent interaction using social integrity constraints. In *LCMAS'03: Logic and Communication in Multi-Agent Systems*, volume 85(2) of *ENTCS*, pages 94–116, Eindhoven, the Netherlands, 2004b.

Marco Alberti, Marco Gavanelli, Evelina Lamma, Federico Chesani, Paola Mello, and Paolo Torroni. Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence*, 20(2–4):133–157, 2006.

Sandra Alves and Mario Florido. Type inference using Constraint Handling Rules. In Hanus [2002], pages 56–72.

T. Andreasen, H. Christiansen, and H.L. Larsen, editors. *FQAS'98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems*, volume 1495 of *LNAI*, Roskilde, Denmark, May 1998. Springer.

K.R. Apt, A.C. Kakas, E. Monfroy, and F. Rossi, editors. *New Trends in Constraints, Joint ERCIM/Compulog Net Workshop, October 1999, Selected papers*, volume 1865 of *LNCS*, Paphos, Cyprus, 2000. Springer. ISBN 3-540-67885-9.

Krzysztof R. Apt and Eric Monfroy. Constraint programming viewed as rule-based programming. *TPLP*, 1(6):713–750, 2001. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068401000072.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. ISBN 0-521-45520-0.

L. Badea, D. Tilivea, and A. Hotaran. Semantic Web Reasoning for Ontology-Based Integration of Resources. In *PPSWR'04: Proc. 2nd Intl. Workshop on Principles And Practice Of Semantic Web Reasoning*, volume 3208 of *LNCS*, pages 61–75, Saint-Malo, France, September 2004. Springer.

Alma Barranco-Mendoza. *Stochastic and heuristic modelling for analysis of the growth of pre-invasive lesions and for a multidisciplinary approach to early cancer diagnosis*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2005.

Maryam Bavarian and Verónica Dahl. Constraint based methods for biological sequence analysis. *J. Universal Computer Science*, 12(11):1500–1520, 2006.

Hariolf Betz. Relating coloured Petri nets to Constraint Handling Rules. In Djelloul et al. [2007a], pages 33–47.

Hariolf Betz and Thom Frühwirth. A linear-logic semantics for Constraint Handling Rules. In Peter van Beek, editor, *CP'05*, volume 3709 of *LNCS*, pages 137–151, Sitges, Spain, October 2005. Springer.

Hariolf Betz and Thom Frühwirth. A linear-logic semantics for Constraint Handling Rules with disjunction. In Djelloul et al. [2007a], pages 17–31.

Stefano Bistarelli, Thom Frühwirth, Michael Marte, and Francesca Rossi. Soft constraint propagation and solving in Constraint Handling Rules. *Computational Intelligence: Special Issue on Preferences in AI and CP*, 20(2):287–307, May 2004.

Mathieu Boespflug. TaiChi:how to check your types with serenity. *The Monad.Reader*, 9: 17–31, November 2007.

Olivier Bouissou. A CHR library for SiLCC. Diplomathesis, November 2004. Tech. Univ. Berlin, Germany.

Sebastian Brand. A note on redundant rules in rule-based constraint programming. In *CSCLP'02: Joint ERCIM/CologNet Intl. Workshop on Constraint Solving and Constraint Logic Programming, Selected papers*, volume 2627 of *LNCS*, pages 279–336, Cork, Ireland, June 2002. Springer.

Sebastian Brand and Eric Monfroy. Deductive generation of constraint propagation rules. In G. Vidal, editor, *RULE'03: 4th Intl. Workshop on Rule-Based Programming*, volume 86(2) of *ENTCS*, pages 45–60, Valencia, Spain, September 2003.

Stéphane Bressan and Cheng Hian Goh. Answering queries in context. In Andreasen et al. [1998], pages 68–82.

Lledó Museros Cabedo and María Teresa Escrig. Modeling motion by the integration of topology and time. *J. Universal Computer Science*, 9(9):1096–1122, 2003.

Wei-Ngan Chin, Martin Sulzmann, and Meng Wang. A type-safe embedding of Constraint Handling Rules into Haskell. Honors Thesis, 2003. School of Computing, National University of Singapore.

Henning Christiansen. CHR grammars. In *TPLP* Abdennadher et al. [2005], pages 467–501.

Henning Christiansen. On the implementation of global abduction. In Katsumi Inoue, Ken Satoh, and Francesca Toni, editors, *CLIMA'06: 7th Intl. Workshop on Computational Logic in Multi-Agent Systems – Revised, Selected and Invited Papers*, volume 4371 of *LNCS*, pages 226–245, Hakodate, Japan, May 2006. Springer.

Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In M. Gabbrielli and G. Gupta, editors, *ICLP'05*, volume 3668 of *LNCS*, pages 159–173, Sitges, Spain, October 2005. Springer.

V. Dahl and I. Niemelä, editors. *ICLP'07: Proc. 23rd Intl. Conf. Logic Programming*, volume 4670 of *LNCS*, Porto, Portugal, September 2007. Springer.

Verónica Dahl and Philippe Blache. Extracting selected phrases through constraint satisfaction. In *Proc. 2nd Intl. Workshop on Constraint Solving and Language Processing*, Sitges, Spain, October 2005.

Verónica Dahl and Baohua Gu. Semantic property grammars for knowledge extraction from biomedical text. In Etalle and Truszczynski [2006], pages 442–443.

Verónica Dahl and Kimberly Voll. Concept formation rules: An executable cognitive model of knowledge construction. In *NLUCS'04: Proc. First Intl. Workshop on Natural Language Understanding and Cognitive Sciences*, Porto, Portugal, April 2004.

Leslie De Koninck. Logical Algorithms meets CHR: A Meta-Complexity Result for Constraint Handling Rules with Rule Priorities. *TPLP*, 9(2):165–212, 2009.

Leslie De Koninck, Tom Schrijvers, and Bart Demoen. INCLP($\mathbb{R}$) - Interval-based nonlinear constraint logic programming over the reals. In Fink et al. [2006], pages 91–100.

B. Demoen and V. Lifschitz, editors. *ICLP'04: Proc. 20th Intl. Conf. Logic Programming*, volume 3132 of *LNCS*, Saint-Malo, France, September 2004. Springer.

K. Djelloul, G. J. Duck, and M. Sulzmann, editors. *CHR'07: Proc. 4th Workshop on Constraint Handling Rules*, Porto, Portugal, September 2007a.

Khalil Djelloul, Thi-Bich-Hanh Dao, and Thom Frühwirth. Toward a first-order extension of Prolog's unification using CHR: a CHR first-order constraint solver over finite or infinite trees. In *SAC'07: Proc. 2007 ACM Symp. Applied computing*, pages 58–64, Seoul, Korea, 2007b. ACM Press. ISBN 1-59593-480-4.

Gregory J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Australia, December 2005.

Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP'03*, pages 79–90, Uppsala, Sweden, 2003. ACM Press. ISBN 1-58113-705-2.

Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Demoen and Lifschitz [2004], pages 90–104.

Gregory J. Duck, Peter J. Stuckey, and Martin Sulzmann. Observable confluence for Constraint Handling Rules. In Dahl and Niemelä [2007], pages 224–239.

María Teresa Escrig and Francisco Toledo. A framework based on CLP extended with CHRs for reasoning with qualitative orientation and positional information. *J. Visual Languages and Computing*, 9(1):81–101, 1998a.

María Teresa Escrig and Francisco Toledo. *Qualitative Spatial Reasoning: Theory and Practice — Application to Robot Navigation*. IOS Press, 1998b.

S. Etalle and M. Truszczynski, editors. *ICLP'06: Proc. 22nd Intl. Conf. Logic Programming*, volume 4079 of *LNCS*, Seattle, Washington, August 2006. Springer.

François Fages, Cleyton Mario de Oliveira Rodrigues, and Thierry Martinez. Modular CHR with ask and tell. In Schrijvers et al. [2008], pages 95–109.

M. Fink, H. Tompits, and S. Woltran, editors. *WLP'06: Proc. 20th Workshop on Logic Programming*, T.U.Wien INFSYS research report 1843-06-02, Vienna, Austria, February 2006.

Aykut Firat. *Information Integration Using Contextual Knowledge and Ontology Merging*. PhD thesis, MIT Sloan School of Management, MA, USA, September 2003.

T. Frühwirth et al., editors. *RCoRP'00: Proc. 1st Workshop on Rule-Based Constraint Reasoning and Programming*, London, UK, July 2000.

Thom Frühwirth. Constraint simplification rules. Technical Report ECRC-92-18, European Computer-Industry Research Centre, Munich, Germany, 1992.

Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming*, 37(1–3):95–138, 1998.

Thom Frühwirth. Proving termination of constraint solver programs. In Apt et al. [2000], pages 298–317. ISBN 3-540-67885-9.

Thom Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In A. Di Pierro and H. Wiklicky, editors, *QAPL'01: Proc. First Intl. Workshop on Quantitative Aspects of Programming Languages*, volume 59(3) of *ENTCS*, Florence, Italy, 2002a.

Thom Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *KR'02: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning*, pages 547–557, Toulouse, France, April 2002b. Morgan Kaufmann.

Thom Frühwirth. Deriving linear-time algorithms from union-find in CHR. In Schrijvers and Frühwirth [2006], pages 49–60.

Thom Frühwirth. Complete propagation rules for lexicographic order constraints over arbitrary domains. In Brahim Hnich, Mats Carlsson, François Fages, and Francesca Rossi, editors, *CSCLP'05: Recent Advances in Constraints, Joint ERCIM/CoLogNET Intl. Workshop on Constraint Solving and CLP, Revised Selected and Invited Papers*, volume 3978 of *LNAI*, Uppsala, Sweden, 2006b. Springer.

Thom Frühwirth. Description logic and rules the CHR way. In Djelloul et al. [2007a], pages 49–61.

Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer, 2003. ISBN 3540676236.

Thom Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic Constraint Handling Rules. In M. Comini and M. Falaschi, editors, *WFLP'02: Proc. 11th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, volume 76 of *ENTCS*, Grado, Italy, June 2002.

Maurizio Gabbrielli, Jacopo Mauro, Maria Chiara Meo, and Jon Sneyers. Decidability Properties for Fragments of CHR. *TPLP*, 10(4-6):611–626, 2010.

Harald Ganzinger and David A. McAllester. Logical algorithms. In Stuckey [2002], pages 209–223.

Sandrine-Dominique Gouraud and Arnaud Gotlieb. Using CHRs to generate functional test cases for the Java card virtual machine. In P. Van Hentenryck, editor, *PADL'06: Proc. 8th Intl. Symp. Practical Aspects of Declarative Languages*, volume 3819 of *LNCS*, pages 1–15, Charleston, SC, USA, January 2006. Springer.

M. Hanus, editor. *WFLP'01: Proc. 10th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, volume 64 of *ENTCS*, Kiel, Germany, November 2002.

Michael Hanus. Adding Constraint Handling Rules to Curry. In Fink et al. [2006], pages 81–90.

C. Holzbaur and Th. Frühwirth, editors. *Special Issue on Constraint Handling Rules*, volume 14(4) of *J. Applied Artificial Intelligence*, April 2000a.

Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. In *J. Applied Artificial Intelligence, Special Issue on Constraint Handling Rules* Holzbaur and Frühwirth [2000a], pages 369–388.

Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. In *TPLP* Abdennadher et al. [2005], pages 503–531.

A. King, editor. *LOPSTR'07: 17th Intl. Symp. Logic-Based Program Synthesis and Transformation, Revised Selected Papers*, volume 4915 of *LNCS*, Kongens Lyngby, Denmark, 2008.

Donald E. Knuth and Peter B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

N. Kobayashi, editor. *APLAS'06: Proc. 4th Asian Symp. on Programming Languages and Systems*, volume 4279 of *LNCS*, Sydney, Australia, November 2006. Springer. ISBN 3-540-48937-1.

Ekkehard Krämer. A generic search engine for a Java Constraint Kit. Diplomarbeit, 2001. Inst. of Comp. Sc., LMU, Munich, Germany.

Edmund S.L. Lam and Martin Sulzmann. Towards agent programming in CHR. In Schrijvers and Frühwirth [2006], pages 17–31.

Edmund S.L. Lam and Martin Sulzmann. A concurrent Constraint Handling Rules semantics and its implementation with software transactional memory. In Neal Glew and Guy E. Blelloch, editors, *DAMP'07: Proc. ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, Nice, France, January 2007.

Johannes Langbein, Frank Raiser, and Thom Frühwirth. A State Equivalence and Confluence Checker for CHR. In P. Van Weert and L. De Koninck, editors, *CHR '10*, pages 1–8. K.U.Leuven, Dept. Comp. Sc., Technical report CW 588, July 2010.

Heiko Lötzbeyer and Alexander Pretschner. AutoFocus on constraint logic programming. In *LPSE'00: Proc. Intl. Workshop on (Constraint) Logic Programming and Software Engineering*, London, United Kingdom, July 2000.

Thierry Martinez. Semantics-Preserving Translations Between Linear Concurrent Constraint Programming and Constraint Handling Rules. In M. Fernández, editor, *PPDP '10*. ACM Press, July 2010.

Jacopo Mauro, Maurizio Gabbrielli, Maria Chiara Meo, and Jon Sneyers. Decidability Properties for Fragments of CHR. *TPLP*, 10(4–6):611–626, 2010.

Marc Meister. *Advances in Constraint Handling Rules*. PhD thesis, Ulm University, Ulm, Germany, 2008.

Marc Meister, Khalil Djelloul, and Thom Frühwirth. Complexity of a CHR solver for existentially quantified conjunctions of equations over trees. In F. Azevedo et al., editors, *CSCLP'06: Proc. 11th Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, volume 4651 of *LNCS*, pages 139–153, Caparica, Portugal, June 2006. Springer. ISBN 978-3-540-73816-9. URL `http://dx.doi.org/10.1007/978-3-540-73817-6_9`.

Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002. ISBN 978-0-387-95250-5.

Paolo Pilozzi and Danny De Schreye. Termination analysis of CHR revisited. In Schrijvers et al. [2008], pages 35–50.

Detlef Plump. *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–213. John Wiley, 1993.

Alexander Pretschner, Oscar Slotosch, Ernst Aiglstorfer, and Stefan Kriebel. Model-based testing for real. *J. Software Tools for Technology Transfer*, 5(2–3):140–157, 2004.

Frank Raiser. *Graph Transformation Systems in Constraint Handling Rules: Improved Methods for Program Analysis*. PhD thesis, Ulm University, November 2010.

Frank Raiser and Thom Frühwirth. Towards term rewriting systems in Constraint Handling Rules machines. In Schrijvers et al. [2008], pages 19–33.

Frank Raiser and Paolo Tacchella. On confluence of non-terminating CHR programs. In Djelloul et al. [2007a], pages 63–76.

Carlos Ribeiro, André Zúquete, Paulo Ferreira, and Paulo Guedes. Security policy consistency. In Frühwirth et al. [2000].

Beata Sarna-Starosta and C.R. Ramakrishnan. Compiling Constraint Handling Rules for efficient tabled evaluation. In M. Hanus, editor, *PADL'07: Proc. 9th Intl. Symp. Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 170–184, Nice, France, January 2007. Springer.

Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *AAAI'07: Proc. 22nd AAAI Conf. Artificial Intelligence*, pages 1191–1196, Vancouver, Canada, July 2007. AAAI Press.

Matthias Schmauß. An implementation of CHR in Java. Diplomarbeit, November 1999. Inst. of Comp. Sc., LMU, Munich, Germany.

T. Schrijvers and Th. Frühwirth, editors. *CHR'05: Proc. 2nd Workshop on Constraint Handling Rules*, K.U.Leuven, Dept. Comp. Sc., Technical report CW421, Sitges, Spain, 2005.

T. Schrijvers and Th. Frühwirth, editors. *CHR'06: Proc. 3rd Workshop on Constraint Handling Rules*, K.U.Leuven, Dept. Comp. Sc., Technical report CW452, Venice, Italy, July 2006.

T. Schrijvers, Th. Frühwirth, and F. Raiser, editors. *CHR'08: Proc. 5th Workshop on Constraint Handling Rules*, Hagenberg, Austria, July 2008.

Tom Schrijvers. Jmmsolve: A generative Java memory model implemented in Prolog and CHR. In Demoen and Lifschitz [2004], pages 475–476.

Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.

Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Th. Frühwirth and M. Meister, editors, *CHR'04, Selected Contributions*, pages 8–12, Ulm, Germany, May 2004.

Tom Schrijvers and Thom Frühwirth. Optimal union-find in Constraint Handling Rules. *TPLP*, 6(1–2):213–224, 2006. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068405002541.

Tom Schrijvers and David S. Warren. Constraint Handling Rules and tabled execution. In Demoen and Lifschitz [2004], pages 120–136.

Tom Schrijvers, David S. Warren, and Bart Demoen. CHR for XSB. In R. Lopes and M. Ferreira, editors, *CICLOPS'03: Proc. 3rd Intl. Colloq. Implementation of Constraint and Logic Programming Systems*, Univ. of Porto, Portugal, Dept. Comp. Sc., Tech. rep. DCC-2003-05, pages 7–20, Mumbai, India, December 2003.

Tom Schrijvers, Jan Wielemaker, and Bart Demoen. Constraint Handling Rules for SWI-Prolog. In A. Wolf, Th. Frühwirth, and M. Meister, editors, *WCLP'05*, volume 2005-01 of *Ulmer Informatik-Berichte*, Universität Ulm, Germany, February 2005.

Tom Schrijvers, Bart Demoen, Gregory J. Duck, Peter J. Stuckey, and Thom Frühwirth. Automatic implication checking for CHR constraints. In *RULE'05: 6th Intl. Workshop on Rule-Based Programming*, volume 147(1) of *ENTCS*, pages 93–111, Nara, Japan, January 2006a.

Tom Schrijvers, Neng-Fa Zhou, and Bart Demoen. Translating Constraint Handling Rules into Action Rules. In Schrijvers and Frühwirth [2006], pages 141–155.

Christian Seitz, Bernhard Bauer, and Michael Berger. Planning and scheduling in multi agent systems using Constraint Handling Rules. In *IC-AI'02: Proc. Intl. Conf. Artificial Intelligence*, Las Vegas, NV, USA, June 2002. CSREA Press.

Jon Sneyers. Turing-complete subclasses of CHR. In María García de la Banda and Enrico Pontelli, editors, *ICLP'08*, LNCS, pages 759–763, Udine, Italy, December 2008a. Springer.

Jon Sneyers. *Optimizing Compilation and Computational Complexity of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, November 2008b.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In Schrijvers and Frühwirth [2005], pages 3–17. URL `http://www.cs.kuleuven.be/~dtai/projects/CHR/biblio/chr2005/sney_schr_demoen_chr_complexity_chr05.ps`.

Jon Sneyers, Peter Van Weert, and Tom Schrijvers. Aggregates for Constraint Handling Rules. In Djelloul et al. [2007a], pages 91–105.

Jon Sneyers, Wannes Meert, and Joost Vennekens. CHRiSM: Chance Rules induce Statistical Models. In F. Raiser and J. Sneyers, editors, *CHR '09*, pages 62–76. K.U.Leuven, Dept. Comp. Sc., Technical report CW 555, July 2009a.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. *ACM TOPLAS*, 31(2):1–42, 2009b.

Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based Probabilistic Logic Learning. *TPLP*, 10(4-6):433–447, 2010a.

Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming*, 10(1):1–47, January 2010b.

P. J. Stuckey, editor. *ICLP'02: Proc. 18th Intl. Conf. Logic Programming*, volume 2401 of *LNCS*, Copenhagen, Denmark, July/August 2002. Springer.

Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM TOPLAS*, 27(6): 1216–1269, 2005. ISSN 0164-0925.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Type processing by constraint reasoning. In Kobayashi [2006], pages 1–25. ISBN 3-540-48937-1. Invited talk.

Martin Sulzmann and Edmund S. L. Lam. Parallel Execution of Multi-set Constraint Rewrite Rules. In S. Antoy and E. Albert, editors, *PPDP '08*, pages 20–31. ACM Press, July 2008.

Martin Sulzmann and Edmund S.L. Lam. Haskell - Join - Rules. In Olaf Chitil, editor, *IFL'07: 19th Intl. Symp. Implementation and Application of Functional Languages*, pages 195–210, Freiburg, Germany, September 2007.

Martin Sulzmann, Tom Schrijvers, and Peter J. Stuckey. Principal type inference for GHC-style multi-parameter type classes. In Kobayashi [2006], pages 26–43. ISBN 3-540-48937-1.

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. Understanding functional dependencies via Constraint Handling Rules. *J. Functional Prog.*, 17(1):83–129, 2007. doi: http://dx.doi.org/10.1017/S0956796806006137.

Robert Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/62.2160.

Michael Thielscher. Reasoning about actions with CHRs and finite domain constraints. In Stuckey [2002], pages 70–84.

Michael Thielscher. FLUX: A logic programming method for reasoning agents. In *TPLP* Abdennadher et al. [2005], pages 533–565.

Peter Van Weert. Efficient Lazy Evaluation of Rule-Based Programs. *IEEE Transactions on Knowledge and Data Engineering*, 99(RapidPosts), 2009.

Peter Van Weert. *Extension and Optimising Compilation of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, May 2010.

Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Schrijvers and Frühwirth [2005], pages 47–62. URL `http://www.cs.kuleuven.be/~dtai/projects/CHR/biblio/chr2005/vanweert_schr_demoen_jchr_chr05.pdf`.

Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In Schrijvers and Frühwirth [2006], pages 125–140.

Peter Van Weert, Jon Sneyers, and Bart Demoen. Aggregates for CHR through program transformation. In King [2008].

Peter Van Weert, Pieter Wuille, Tom Schrijvers, and Bart Demoen. CHR for imperative host languages. In Tom Schrijvers and Thom Frühwirth, editors, *Constraint Handling Rules — Current Research Topics*, volume 5388 of *LNAI*, pages 161–212. Springer, December 2008.

Dean Voets, Paolo Pilozzi, and Danny De Schreye. A new approach to termination analysis of Constraint Handling Rules. In Djelloul et al. [2007a], pages 77–89.

Kimberly Voll. *A methodology of error detection: Improving speech recognition in radiology.* PhD thesis, Simon Fraser University, Burnaby, Canada, 2006.

Armin Wolf. *Adaptive Constraintverarbeitung mit Constraint-Handling-Rules – Ein allgemeiner Ansatz zur Lösung dynamischer Constraint-probleme.* PhD thesis, Tech. Univ. Berlin, Germany, 1999.

Armin Wolf. Projection in adaptive constraint handling. In Apt et al. [2000], pages 318–338. ISBN 3-540-67885-9.

Armin Wolf. Toward a rule-based solution of dynamic constraint hierarchies over finite domains. In Frühwirth et al. [2000].

Armin Wolf. Adaptive constraint handling with CHR in Java. In T. Walsh, editor, *CP'01*, volume 2239 of *LNCS*, pages 256–270, Paphos, Cyprus, 2001a. Springer. ISBN 3-540-42863-1.

Armin Wolf. Attributed variables for dynamic constraint solving. In *Proc. 14th Intl. Conf. Applications of Prolog*, pages 211–219, Tokyo, Japan, October 2001b.

Armin Wolf. Intelligent search strategies based on adaptive Constraint Handling Rules. In *TPLP* Abdennadher et al. [2005], pages 567–594.

Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On incremental adaptation of CHR derivations. In *J. Applied Artificial Intelligence, Special Issue on Constraint Handling Rules* Holzbaur and Frühwirth [2000a], pages 389–416.

Pieter Wuille, Tom Schrijvers, and Bart Demoen. CCHR: the fastest CHR implementation, in C. In Djelloul et al. [2007a], pages 123–137.

Part II

Implementation and Optimization of CHR

Chapter 2

Basic Compilation

Author:	Gregory J.Duck
Thesis Title:	Compilation of Constraint Handling Rules
School:	University of Melbourne, Australia
Publication Year:	2005

Foreword

Over the years, many different CHR compilers have been implemented in a variety of programming languages and programming language paradigms. Currently there are CHR compilers for constraint logic programming languages such as Prolog and Mercury, functional languages such as Haskell, and even in more traditional imperative languages such as C and Java. Compilers range from the "state-of-the-art" such as the K.U.Leuven CHR system [Schrijvers and Demoen, 2004], which implements powerful program analysis and optimization, to very simple compilers such as `toychr` [Duck], which is implemented in a few hundred lines of Prolog code.

This chapter focuses on the very basic compilation of CHR into a logic programming language similar to Prolog and Mercury. The basic compilation is useful for two reasons. Firstly, it can be relatively easily adapted to other programming languages that currently have no CHR implementation. Secondly, the basic compilation serves as a foundation for more advanced compilation techniques including program analysis and optimization. Optimizing CHR compilers use essentially the same compilation schema as the basic version presented in this chapter, adapted appropriately.

This chapter assumes the reader is familiar with the refined operational semantics [Duck et al., 2004] (or simply the "refined semantics") of CHR. The refined semantics was originally a formalisation of the operational semantics of CHR implementations of the time, and is more restrictive than the *theoretical* operational semantics for CHR. Since then most CHR compilers respect some variant of refined semantics. It has become an unofficial standard for CHR implementations. Because the refined semantics closely matches the call-based semantics of most compiler target languages, the compilation schema is simple.

This chapter also assumes the reader is relatively familiar with logic programming languages. The code examples also include type and mode declarations that are featured in the Mercury language. The original version of this chapter used the HAL [Demoen et al., 1999] constraint logic programming language as the target language. Since then the HAL language has become extinct. In HAL's place we use a Mercury-like logic programming language.

2.1 Introduction

In this chapter we explain the *basic compilation* of CHRs into logic programming languages such as Prolog or Mercury [Somogyi et al., 1995]. We present all of the information required to make a simple "no-frills" CHR compiler that performances reasonably well for many programs. This will form the basis for more advanced compilation, including optimisation.

The specification of the refined operational semantics [Duck et al., 2004] describes a state machine with (reasonably complicated) transitions between states. It is straightforward to implement a naive interpreter for this state machine. Interpretation is generally slower than compilation, since much of the specification is implemented manually, for instance the execution stack. For compiled CHRs, large chuncks of the functionality required may already be provided by the target language, e.g. the execution stack becomes the call stack, etc.

Compilation of CHRs is very similar to compiling other programming languages in that it is a multi-phase process: The first phase is parsing and normalisation (desugaring), followed by analysis, optimisation and then finally code generation. Usually analysis and optimisation are optional, and are not covered here. We will briefly look at parsing and program normalisation, but the main focus of this chapter will be on code generation.

Ideally the output of the CHR compiler should be as efficient or better than the code a human would write.

Example 1. *Consider the classic CHR* `gcd` *program.*

```
gcd(0) <=> true.
gcd(M) \ gcd(N) <=> M =< N | gcd(M-N).
```

The following is a similar program written in Prolog by a human. It is provided as a benchmark for comparison later on.

```
gcd(N,M,R) :-
     ( M =< N ->
        ( M = 0 ->
           R = N
        ;  gcd(N-M,M,R)
        )
     ;  gcd(M,N,R)
     ).
```

□

Unfortunately the result of the basic compilation of `gcd` will be very different than the human implemented version above.

The rest of this chapter is divided up as follows. First we will briefly look at parsing and program normalisation in section 2.2. Section 2.3 will describe our simple runtime system in preparation for Section 2.4, which describes code generation. Next we devote Section 2.5 to the surprisingly tricky problem of compiling guards. Finally we conclude.

2.2 Parsing and Normalisation

Parsing and normalisation are the first phases in any compilation process. We briefly describe parsing and normalisation of CHR programs.

2.2.1 Parsing

The general problem of parsing any programming language, including CHRs, is well-studied and there are many tools that help automate this process. Since CHRs are usually embedded in a host language (in this case a logic programming language), the host language's parser is adapted to recognise CHRs.

Assuming that there are no syntax errors, the result of the parsing is usually a list of some representation of the rules in the program. For example, we can represent the rule $(r \ @ \ H_1 \setminus H_2 \iff g \mid B)$ as the term `rule(`r`,`H_1`,`H_2`,`g`,`B`)`. We also translate the head, guard and body into lists of terms, where each term represents a constraint from the original rule. The rule name r is optional, so if it is omitted by the programmer usually the parser will generate one (making sure it is unique with respect to the other rules).

Example 2. *Consider the* `gcd` *program from Example 1. After parsing the program under our scheme the result is the following list of rules.*

```
[ rule(gcd1,[],[gcd(0)],[],[true]),
  rule(gcd2,[gcd(M)],[gcd(N)],[M =< N],[gcd(M-N)])]
```

□

2.2.2 Head and Guard Normalisation

Normalisation is a preprocessing step aimed at making all guards explicit, since non-variable terms and matching variables appearing in the head of a rule or guard can be represented by guards instead. *Head normalisation* is achieved by iteratively applying the following steps

1. Rewrite each constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ where c is an n-ary constraint symbol and t_i is a non-variable term, to $c(t_1, \ldots, X, \ldots, t_n)$ and add the constraint $X = t_i$ to the guard.

2. If variable X appears as an argument more than once in the head of a rule, replace one occurrence with a new variable, say X', and add the constraint $X = X'$ to the guard.

After normalisation, each head simply provides the multiset of names of constraints which can match that rule, while the guard indicates which such multisets actually match.

To simplify analysis and compilation, guards are also normalised.

1. Rewrite each constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ where c is an n-ary constraint symbol and t_i is either a non-variable term or a variable which appears as an argument elsewhere in the guard, to $X' = t_i \wedge c(t_1, \ldots, X', \ldots, t_n)$ where X' is a new variable.

2. Rewrite each equation $X = f(t_1, \ldots, t_i, \ldots, t_n)$ where f is an n-ary function and t_i is either a non-variable term or a variable which appears as an argument elsewhere in the guard, to $X' = t_i \wedge X = f(t_1, \ldots, X', \ldots, t_n)$, where X' is a new variable.

3. Add explicit existential quantification for each existentially quantified variable in the guard.

In CHR, a variable that appears in the guard but not in the rule head is implicitly existentially quantified. For explicit existential quantification we introduce the notation

`exists [`$X_1, ..., X_n$`] (`g`)`

to indicate that variables $X_1, ..., X_n$ are existentially quantified in guard g.

After guard normalisation, each constraint is either an equation of the form $X = Y$, $X = f(Y_1, ..., Y_n)$, or a constraint $c(Y_1, ..., Y_n)$ where $X, Y, Y_1, ..., Y_n$ are all distinct variables. Both head and guard normalisation preserve the operational and declarative meanings of the program. Note that sometimes multiple normalisations are possible, e.g. the head `leq(`X`,`X`)` can be normalised as `leq(`X`,`N`)` or `leq(`N`,`X`)` with guard $X = N$. Such normalisations are always unique up to variable renaming, hence any normalisation can be used.

Example 3. *Consider the following CHR program defining a* ***`length(Xs,L)`*** *constraint which holds if* `L` *is the length of list* `Xs`*. Unlike the standard* ***`length`****/2 predicate, this version works in any mode, including when both arguments are fresh variables.*

```
length([],L)     <=> L = 0.
length(Xs,0)     <=> Xs = [].
length([_|Xs],L) <=> L = L1+1, length(Xs,L1).
length(Xs,L)     <=> 1 =< L | Xs = [_|Ys], length(Ys,L-1).
length(_,L)      ==> 0 =< L.
```

The following is the same program after head and guard normalisation has been applied.

```
length(Xs,L)  <=> Xs = []                       | L = 0.
length(Xs,L)  <=> L = 0                         | Xs = [].
length(Xs1,L) <=> exists [A,Xs] (Xs1 = [A|Xs]) |
                        L = L1+1, length(Xs,L1).
length(Xs,L)  <=> exists [L1] (L1 = 0, L > L1) |
                        Xs = [_|Ys], length(Ys,L-1).
length(_,L)   ==> true                          | L >= 0.
```

Thanks to head normalisation, all variables appearing in the rule heads are distinct variables. Guard normalisation has made all implicit existential quantification explicit and reduced the guard into the simplified form. □

2.2.3 Program Normalisation

Our representation still closely resembles the original program, but this is not very helpful for the later phases of compilation. Since the refined operational semantics treats a constraint as a call, and checks each occurrence in order, it is useful to create a mapping between the constraint and a list of occurrences for that constraint. This mapping is the normal form of our program.

First we must define a data structure to represent an individual occurrence for a constraint. Suppose we have a rule `rule(`r`,`H_1 ++ $[c]$ ++ H_2`,`H_3`,`g`,`B`)`, then our representation of the occurrence in that rule for c is `occ(`c`,remain,`n`,`H_1 ++ H_2`,`H_3`,`g`,`B`,`r`)`. The first field contains the constraint from the head that must match the active constraint. The second field is the constant `remain`, which represents the fact that this occurrence does not delete the active constraint (in the other case, this field will contain the constant `delete`). The third field n is the occurrence number, which must be calculated during normalisation. For example, if the occurrence of c is the third in the program, then $n = 3$. The fourth field contains the non-deleted part of the head. The rest of the fields, e.g. H_3, g, etc., are exactly the same as in the original rule.

The other case is where the occurrence for c is in the deleted part of the head, i.e. `rule(`r`,`H_1`,`H_2 ++ [c] ++ H_3`,`g`,`B`)`, then the representation of the occurrence is `occ(`c`,delete,`n`,`H_1`,`H_2 ++ H_3`,`g`,`B`,`r`)`.

Example 4. *The list of occurrences for constraint* `gcd`/1 *from Example 2 is the following.*

```
[ occ(gcd(X),delete,1,[],[],[X = 0],[true],gcd1),
  occ(gcd(N),delete,2,[gcd(M)],[],[M =< N],[gcd(M-N)],gcd2),
  occ(gcd(M),remain,3,[],[gcd(N)],[M =< N],[gcd(M-N)],gcd2)]
```

Notice that the head of the first occurrence has been normalised. □

The advantage of the normal form this that all of the information required to compile an individual occurrence is now in one place.

2.3 Runtime Environment

The refined operational semantics defines an *execution state* to be a tuple containing an execution stack, CHR store, built-in store and propagation history. In this section we show how to implement each of these in a pure logic programming language with minimal extensions.

2.3.1 Execution Stack

The refined operational semantics [Duck et al., 2004] of CHRs explicitly represent the execution stack as a sequence of constraints, numbered constraints and active constraints. In practice the execution stack is nothing more than the ordinary program call stack in Mercury or Prolog.

The execution stack starts off as a sequence of (non-numbered) constraints and built-in constraints. Built-in constraints are not treated specially in any way, they are just ordinary procedure calls. CHR constraints are different in the sense that the CHR compiler must generate the required code. Given a CHR constraint in the original CHR program, the CHR compiler generates a predicate, which implements the constraint, with exactly the same interface as the original. The predicate that the compiler generates is called the *top-level predicate* and will be explained in the code generation section.

The execution stack also contains active constraints. An active constraint of the form $p(X_1, ..., X_n)\#I : m$ will be implemented by an *occurrence predicate* with interface `p_`m`(I,`X_1`,...,`X_n`)`. The occurrence predicate will contain the code generated by the compiler for finding matches for the occurrence m of constraint `p`. Occurrence predicates are also *chained*, i.e. `p_`m calls `p_`$(m + 1)$ if the rule cannot fire (i.e. the **Default** transition). Exactly how this is done is left for the code generation section.

Finally, the execution stack contains numbered constraints, which are woken up from the store after a **Solve** transition. In the implementation the **Solve** and **Reactivate** transitions are implemented together, so there is no special representation of numbered constraints on the stack, only active constraints.

2.3.2 CHR Store

CHR constraints in the store are of the form $c\#i$ where i a unique number. A naive implementation of constraint identifiers could use ordinary integers (as in the specification of the refined semantics), however there are reasons why this is an inefficient approach.

The operation of testing if a constraint identifier belongs to a deleted constraint turns out to be a useful and common operation. While an ordinary integer has no memory of if it has been deleted or not, a cleverer implementation is to use a fresh Herbrand variable as a constraint identifier. The advantage is that when a constraint is deleted, the variable identifier is bound to some pre-defined atom, e.g. '`deleted`', and then testing if a CHR constraint has been deleted is reduced to testing if the identifier is still a variable (a very fast operation in Prolog). This approach was first used by the ECL^{i}PS^e CHR compiler [Frühwirth and Brisset, 1995].

A Mercury implementation of constraint identifiers is similar except that instead of using variables a special mutable[1] data structure is used. This data structure has two states: either the constraint identifier belongs to a deleted constraint or not. We refer to an identifier belonging to a deleted constraint as a *dead* identifier, otherwise it is *alive*.

We define the following operations on constraint identifiers.

- `new(Id)` – creates a new constraint identifier `Id`;
- `kill(Id)` – marks the constraint identifier `Id` as being dead; and
- `alive(Id)` – succeeds if `Id` is alive, otherwise fails.

In the case of the fresh variable implementation, `new(Id)` is equivalent to a NOP (No OPeration), as this implicitly creates a fresh variable, and `alive(Id)` is equivalent to Prolog's `var(Id)` which succeeds if `Id` is a variable.

A numbered constraint in the CHR store will be represented as the special tuple c `#` i, where c is the constraint and i is the identifier. Our representation of the global CHR store will be a single global list for simplicity. Searching for matching constraints against some rule will involve iterating through this global list. The cost of this simplicity is that searching for matching partners in a list is an $O(N)$ operation, where N is the length of the list.

The CHR store is usually implemented by global variables, which are supported by Mercury and some implementations of Prolog. We assume the following abstract operations on the global store.

- `insert(C,Id)` – Insert constraint `C # Id` into the global CHR store;
- `delete(Id)` – Delete the constraint associated with identifier `Id` from the global CHR store and mark `Id` as *dead*; and
- `get_iterator(Ls)` – Binds `Ls` to be a list of all constraints in the global CHR store;

The `get_iterator` operation will be used by the code for finding matchings.

One obvious improvement is to specialise `get_iterator` for each of the predicate symbols of the constraints in the program. For example, if the program defines a constraint `p`/3, then a specialised `get_iterator_p_3` returns an iterator containing only `p`/3 constraints. For simplicity, we will stick to the single global list, and leave more advanced schemes for optimising compilation.

2.3.3 Built-in Store

CHR programs may extend zero or more built-in solvers. In all current Prolog implementations of CHRs it is always assumed that there is exactly one built-in solver: the Herbrand

[1] Changes to mutable data structures are trailed, so they will be undone on backtracking.

equation solver (i.e. ordinary Prolog unification). In general any number of different built-in solvers are possible, but for an implementation, there must be communication between compiled CHR code and these solvers.

Example 5 (Length Constraint)**.** *Consider the* `length` *program from Example 3 which extends both Herbrand and finite domain solvers. Consider the goal* `length(Xs,L), L = 1`*. Its execution will first add the CHR constraint* `length(Xs,L)` *to the store. This constraint cannot by itself cause the application of any of the rules except rule* (5) *which adds the additional constraint* $L \geq 0$*. Next we add the finite domain constraint* $L = 1$ *to the finite domain solver store. This affects rule* (4) *which can now be applied:* `Xs` *is bound to* `[_|Ys]` *and CHR constraint* `length(Xs,L)` *is replaced by* `length(Ys,L1)` *where* $L1 = L - 1 = 0$*. The second rule simplifies* `length(Ys,L1)` *to the Herbrand constraint* `Ys = []`*. Hence, the final solution is* `Xs = [_], L = 1`*.* □

We can see three kinds of interaction between the CHR solver and the built-in solvers in the example above.

1. The CHR solver adds new constraints to the built-in solvers.

2. The CHR solver asks the built-in solvers whether constraints are *entailed*. This is for testing the guard holds in the **Simplify** and **Propagate** transitions under the refined semantics.

3. The built-in solvers must alert the CHR solver whenever non-trivial changes to the built-in store occur. This is to correctly implement the **Solve** transition, which requires all non-ground CHR constraints to be woken up.

Constraint solvers, by definition, provide methods that allow new constraints to be added to their store. The second kind of interaction that needs a well defined interface if we wish to extend an arbitrary built-in solver. We will defer consideration of this until Section 2.5.

For the third type of interaction we use a very simple form of dynamic scheduling. We assume the existence of a special predicate `delay`(*Term*,*Id*,*Goal*), which delays *Goal* on the condition that any variable in *Term* has *changed* provided *Id*, which is a constraint identifier, is still alive. Thus, with the appropriate delayed goals set, constraints from the CHR store are in effect re-added to the execution stack each time a constraint is added to the built-in store.

It is usually the solver writer's responsibility to implement the `delay` predicate, because the implementation requires intimate knowledge of the inner workings of that solver. If there are multiple solvers then we assume that `delay`/3 is overloaded. In Prolog the `delay` predicate can be implemented in terms of existing dynamic scheduling constructs, e.g. with attributed variables [Holzbaur, 1992].

2.3.4 Propagation History

The propagation history is very easy to implement naively, but quite challenging to implement efficiently. A naive implementation uses some efficient queryable data structure (e.g. balanced tree or hash table) over the entries. The advantage is that testing if an entry is in the propagation history is very fast, however as program execution proceeds, the propagation history grows in size. Ideally, whenever a constraint is deleted, any entry in the propagation history which contains the corresponding number should also be deleted. We need a mechanism for determining all entries that are associated with a given identifier.

```
:- pred p(t_1,...,t_n).                                    (1)
:- mode p(m_1,...,m_n) is d.                               (2)
p(X_1,...,X_n) :-                                          (3)
    new(Id),                                               (4)
    C = p(X_1,...,X_n),                                    (5)
    insert(C,Id),                                          (6)
    delay(C,Id,p_1(Id,X_1,...,X_n)),                       (7)
    p_1(Id,X_1,...,X_n).                                   (8)
```

Figure 2.1: Pseudo code for a top-level predicate.

We will assume a global propagation history, and all interactions with it go through one operation: `check_history(`*Entry*`)` which fails if *Entry* (which is a list of constraint identifiers and the rule name) is already in the global propagation history, or adds it otherwise. In other words, for a unique new entry *Entry*, the first call to `check_history(`*Entry*`)` will succeed, but all subsequent calls with the same *Entry* will fail. Note that for simplicity, the rule name is encoded as a constraint identifier.

2.4 Code Generation

Code generation is the final stage in any compilation process. In this section we give pseudo code and describe what exactly needs to be generated. The only exception is the implementation of the guard, which is covered by the next section.

2.4.1 Top-level Predicate

The top-level predicate is called in place of the original CHR constraint after compilation. Its role is to perform the necessary initialisation, i.e. allocating a new constraint identifier, insertion into the CHR store and setting up appropriate delayed goals on any free variables. Naturally the top-level predicate has the same interface as the original CHR constraint.

For CHR constraint of functor/arity `p`/n, Figure 2.1 shows the corresponding top-level predicate the CHR compiler generates. Lines (1)-(2) show the `pred` and `mode` declarations. Here $t_1, ..., t_n$, $m_1, ..., m_n$ and d are exactly the argument types, argument modes and determinism for the original CHR constraint `p`/n. For a Prolog implementation these declarations will be omitted. The interface, shown in line (3), is a list of distinct variables $\texttt{X}_1\texttt{,...,X}_n$ which are the arguments to the CHR constraint. Line (4) allocates a new constraint identifier. Line (5) constructs the constraint `C` ready for insertion into the store, whereas line (6) actually does the insertion. Line (7) sets up any necessary delayed goals on any solver variables in `C`. Notice that the same identifier `Id` is used for setting up the delayed goals and as the constraint identifier. Finally, the constraint becomes active in line (8) by calling the first occurrence predicate `p_1`.

2.4.2 Occurrence Predicates

The main purpose of an occurrence predicate is to implement the behaviour of the **Simplify** or **Propagate** transitions from the refined operational semantics. For all CHR constraints `p`/n from the CHR program, and all occurrences m of `p`/n, the CHR compiler generates an occurrence predicate by the name `p_`m. This is usually the bulk of the code generated by the compiler.

```
:- pred p_i(id,t_1,...,t_n).                                    (1)
:- mode p_i(in,m_1,...,m_n) is d.                               (2)
p_i(Id,X_1,...,X_n) :-                                          (3)
    <find-matches-and-call-body>                                (4)
    ( alive(Id) ->                                              (5)
        p_(i+1)(Id,X_1,...,X_n)                                 (6)
    ;   true                                                    (7)
    ).
```

Figure 2.2: Pseudo code for the i^{th} occurrence predicate.

The skeleton of an occurrence predicate (for the i^{th} occurrence) is shown in Figure 2.2. Lines (1)-(2) show the `pred` and `mode` declarations, which are identical to the same declarations for the top-level predicate, except for an extra argument for the constraint identifier (which we assume has type '`id`'). Line (3) is the interface, where the first argument is the constraint's identifier, followed by the constraint's actual arguments. Note that all of $\mathtt{X}_1,\ldots,\mathtt{X}_n$ are the same set of distinct variables (because of normalisation) from the original rule. Line (4) represents the main purpose of this predicate, which is to find matching partners and fire the rule. We leave this part for now. After line (4) we can assume that all possible matchings have been tried. The rest of the occurrence predicate, lines (5)-(7) decides whether or not the next occurrence predicate, $\mathtt{p\_}(i+1)$, should be called. Line (5) tests if the active constraint has been deleted. If not then the next occurrence predicate is called in line (6) (in effect we are applying **Default** to the active constraint), otherwise the active constraint is effectively **Drop**ped in line (7). Note that for the last occurrence lines (5)-(7) are omitted.

The rest of this section is concerned with compiling the join, and firing the rule. Thanks to head and guard normalisation the head of the rule contains CHR constraints with only variable arguments, and none of those variables are repeated anywhere else in the head. This simplifies the problem of finding partner constraints considerably, since all constraints in the store with the same functor/arity as the partner constraint will always potentially match. The job of selecting which of these matches are valid is now entirely decided by the guard. The disadvantage of this approach is efficiency, however a better implementation is beyond the scope of our basic compilation.

Because of program normalisation, the CHR compiler has a list of partner constraints for a given active constraint and occurrence. We can break down the problem of finding a match (for the rule) into the problem of finding a match for individual partner constraints. Basically, given an iterator, we iterate through all potential matches for the first partner, and if a potential match is found, then we iterate through all potential matches for the second partner, and so on. If all partners have been matched, then the rule may fire provided the guard and propagation history tests succeed. After the rule fires, or after it failed to fire (e.g. if the guard failed), we return to the iterator for the last partner to find a new match. Either we find a new match for the last partner, or we return to the iterator for the previous partner, and so on. This system of iteration avoids the need to start the search for a set of matching constraints from scratch each time the rule fires, hence we avoid redundant work.

The code generated by the compiler that attempts to find a match for an individual partner is contained within a special *join-loop* predicate. Basically a join-loop predicate iterates through all potential matches for some partner. If/when a matching is found,

we either call the next join-loop predicate if there are more partners, or we call the *call-body* predicate which does the final checks before firing the rule. Likewise, execution only returns to a join-loop predicate once all matches for the remainder of the join have been tried.

For the implementation of a join-loop predicate there are complications to consider. Firstly, the operational semantics of CHRs disallows the same constraint from the store to be matched against more than one partner.[2] Secondly, the partner constraints that we select from the iterator may have been deleted since the original call to `get_iterator`. The problem arises because `get_iterator` is called once, then we iterate through the list, possibly firing the rule as we go. Unfortunately, it is possible that whilst firing the rule we "delete" some of the constraints in our iterator. Note that deletion removes the constraint from the global CHR store, and not from any (local) iterators, which are in effect a copy of an older store. Similarly, it is possible that any constraint from our partial match has been deleted in a similar fashion.

Since deletion is a problem with iterators, it seems that insertion may also be a problem, i.e. whilst firing a rule we create new constraints that are potential matches. In fact there is no problem because of the call-based behaviour of the refined operational semantics. Any new CHR constraints created by a rule must have finished being active before we return from the firing of the rule. This means that all matches which include the new constraint must have already been tried, so there is no need to update the iterators. Actually this is another advantage of using iterators, since we never consider these constraints we save redundant work.

The pseudo code for a join-loop predicate is shown in Figure 2.3. Here we assume that for an active `p` constraint (at the i^{th} occurrence) we are trying to find a matching partner `q`(A, B, C) (the code is similar for any other constraint). Lines (1)-(2) show the `pred` and `mode` declarations. Line (3) handles the case of an empty iterator, where no action is necessary. Line (4) handles the other case, and also demonstrates the interface for a join-loop predicate. The first argument is the iterator itself, which we assume has type '`iterator`' defined as follows:

```
:- type iterator == list(numbered).
:- type numbered ---> constraint # id.
```

This means an iterator is a list of numbered constraints, where type '`constraint`' is the type given to the constraints in the CHR store. Next is `Id`$_1$`,...,Id`$_n$ which are the constraint identifiers from the active constraint and from any matches of partners so far (the partial match). Similarly, `X`$_1$`,...,X`$_k$ are the combined arguments from the partial match. Here we assume that $t_1, ..., t_k$ and $m_1, ..., m_k$ are the types and modes associated with `X`$_1$`,...,X`$_k$ (based on the types and modes of the constraints involved in the partial match). In line (4) we have also extracted a new constraint `C` with identifier `Id` from the iterator.

The rest of the code is split into two parts. The first part, lines (5)-(10), decides if the constraint `C` matches against the partner (i.e. if `C` is a `q`/3 constraint), and takes appropriate action if so. Line (5) deconstructs the newly acquired constraint `C`, and fails if it is not a `q`/3. Line (6) tests if the identifier `Id` has not been deleted (recall that deletion is possible). Lines (7)-(8) test if the new identifier `Id` has not been matched earlier in this join (this is to ensure multiset matching). If all of these tests pass, then in line (9) we call the code for finding a match for the next partner if necessary, or the call the call-body predicate otherwise.

[2]This was the purpose of the multiset union $\uplus$ in the specification of **Simplify** and **Propagate**.

```
:- pred p_i_join_loop(iterator,id,...,id,t_1,...,t_k).          (1)
:- mode p_i_join_loop(in      ,in,...,in,m_1,...,m_k) is d.    (2)
p_i_join_loop([],_,...,_).                                     (3)
p_i_join_loop([C # Id|Ls],Id_1,...,Id_n,X_1,...,X_k) :-        (4)
    (
      C = q(A,B,C),                                            (5)
      alive(Id),                                               (6)
      Id \= Id_1,                                              (7)
      ⋮
      Id \= Id_n ->                                            (8)
        <find-matches-and-call-body>                           (9)
    ;   true                                                   (10)
    ),
    (
      alive(Id_1),                                             (11)
      ⋮
      alive(Id_n) ->                                           (12)
        p_i_join_loop(Ls,Id_1,...,Id_n,X_1,...,X_k)            (13)
    ;   true                                                   (14)
    ).
```

Figure 2.3: Pseudo code for the join loop predicate.

The second part of the join-loop predicate, in lines (11)-(14), decides if we are allowed to continue looking for partners, or if we must abort because a constraint from our partial match has been deleted. The invariant we are maintaining is that all constraints in the partial match have not been deleted. Lines (11)-(12) test if all the identifiers from the partial matching are still alive. If so then we recursively call the join-loop predicate with the same arguments except with the tail of the original iterator, otherwise no action is necessary.

We can now give a simple example of a join-loop predicate.

Example 6. *Consider occurrence number six for the* `leq` *constraint in the following (normalised) rule.*

```
leq(X,Y)_6, leq(W,Z) ==> Y = W | leq(X,Z).
```

To find matching constraints for the partner `leq(W,Z)` *the compiler generates the join-loop predicate as shown in Figure 2.4. Here, we assume that the type of the arguments to the* `leq` *constraints is* `hint` *(Herbrand* `int`*).* □

After a set of matching constraints has been found we need to check the guard and propagation history, then delete any constraints if necessary, before calling the code in the body. This is the role of the *body-call* predicate, whose pseudo code is shown in Figure 2.5. Lines (1)-(2) are the `pred` and `mode` declarations. The interface to the body call predicate, shown by line (3), is very similar to that of the join-loop predicate. All of $\texttt{Id}_1,\ldots,\texttt{Id}_n$ are constraint identifiers, and all of $\texttt{X}_1,\ldots,\texttt{X}_k$ are arguments of the matching respectively. Note that the interface could be improved by eliminating arguments that are not actually required by the predicate's body.

```
:- pred leq_6_join_loop(iterator,id,hint,hint).
:- mode leq_6_join_loop(in,in,in,in) is semidet.
leq_6_join_loop([],_,_,_).
leq_6_join_loop([C#J|Ls],I,X,Y) :-
    (
      C = leq(W,Z),
      alive(J),
      J \= I ->
        leq_6_call_body(I,J,X,Y,W,Z)
    ;   true
    ),
    (
      alive(I) ->
        leq_6_join_loop(Ls,I,X,Y)
    ;   true
    ).
```

Figure 2.4: Join loop predicate for the transitivity rule's first occurrence.

Line (4) tests the guard and line (5) checks the propagation history. Exactly how the guard is tested is left for the next section. Checking the propagation history involves constructing an entry and then calling `check_history` described previously. The history check is omitted whenever rule r is not a propagation rule. The order of the identifiers in the history entry is defined by the auxiliary function *order*, which is evaluated at compile time. We assume that function *order* sorts the identifiers based on the textual order of the constraints in the rule head. It is important that all call-body predicates use the same order for the same rule.

If both the guard test and history check pass, then the rule can fire. If the rule is not a propagation rule then some of the matching constraints need to be deleted. Let $\{\mathtt{Id}'_1,\ldots,\mathtt{Id}'_j\} \subseteq \{\mathtt{Id}_1,\ldots,\mathtt{Id}_n\}$ be the identifiers of these constraints, then lines (6)-(7) explicitly do the deletions. Finally, the body of the rule is called in line (8). The body is usually copied verbatim from the original rule.

We are now ready for a complete example.

Example 7 (Compiled `gcd` Program)**.** *The compiled version of the* `gcd` *program is given in Figure 2.6. Note that we gloss over guard compilation (the guards are inserted "as-is" into the compiled code) for the time being. Also, we omit the* `pred` *and* `mode` *declarations for brevity. The new program consists of one top-level predicate, three occurrence predicates, two join-loop predicates and three body-call predicates.* □

The compiled version of the `gcd` program is much larger than the original CHR version, and this is generally true for all CHR compilation. Several improvements are possible, e.g. inlining the body-call predicates, and inlining the first occurrence predicate with the top-level predicate. After optimisation, the resulting code will be considerably more compact.

2.5 Compiling the Guard

Let r be a normalised CHR rule with guard g. The operational semantics of CHRs dictates that (a renamed copy of) r can only fire iff $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists_r(\theta \wedge g)$ holds, where B is the

```
:- pred p_i_call_body(id,...,id,t_1,...,t_k).                    (1)
:- mode p_i_call_body(in,...,in,m_1,...,m_k) is d.               (2)
p_i_call_body(Id_1,...,Id_n,X_1,...,X_k) :-                      (3)
    (
      <test-guard>,                                              (4)
      check_history([order(Id_1,...,Id_n),r]) ->                 (5)
        delete(Id'_1),                                           (6)
        ⋮
        delete(Id'_j),                                           (7)
        <body>                                                   (8)
    ;   true
    ).
```

Figure 2.5: Pseudo code for the call body predicate.

current built-in store, θ is the matching substitution, and the (possibly incomplete) test ($\mathcal{D} \models_{\mathcal{S}}$) represents the solver.

In practice the built-in solver must not only provide a procedure for *telling* a constraint (adding it to the built-in store) whenever it appears in the body of the rule, but also a procedure for *asking* a constraint (determining if the guard is entailed by the current built-in store) whenever it appears in the guard of the rule. Formally an ask constraint c holds iff $\mathcal{D} \models_{\mathcal{S}} B \rightarrow c$. For example, in the case of the Herbrand solver, the only tell constraint (`=/2`) has the known associated ask constraint `==/2`.

2.5.1 Basic Guards

This section presents how guard entailment would be implemented in a Mercury CHR compiler. We will first assume that the guard g contains no existentially quantified variables (i.e., all variables in the guard also appear in the head).

In order for a constraint solver to be extended by CHRs, the solver needs to provide *ask* versions of the constraints that it supports. It is the ask version of the constraints that should be used in guards.

Example 8. *For example, consider the following CHR program that implements a* `min`/3 *constraint over a finite domain solver.*

```
min(A,B,C) <=> A =< B | C = A.
min(A,B,C) <=> A >= B | C = B.
```

Consider the compilation of the first rule. The constraint `A =< B` *will be replaced by some predicate which implements the ask version of the finite domain* `=<` *constraint, e.g,* `'ask_=<'(A,B)`. □

CHR implementations typically automatically translate guard tell constraints into ask constraints. However, most implementations only deal with one built-in solver (Herbrand), making the translation trivial (i.e. replace `=` with `==`). When arbitrary solvers are used, the compiler needs a general method for determining the relationship between the tell and ask versions of each constraint so that it can automatically transform one into the other. This can be achieved by implementing a special declaration that defines the relationship between ask and tell constraints, e.g.,

```
gcd(X) :-
    C = gcd(X),
    new(I),
    insert(C,I),
    delay(C,I,gcd_1(I,X)),
    gcd_1(I,X).

gcd_1(I,X) :-
    gcd_1_call_body(I,X),
    ( alive(I) ->
        gcd_2(I,X)
    ;   true
    ).

gcd_1_call_body(I,X) :-
    ( X = 0 ->
        delete(I)
    ;   true
    ).

gcd_2(I,N) :-
    get_iterator(Ls),
    gcd_2_join_loop(Ls,I,N),
    ( alive(I) ->
        gcd_3(I,X)
    ;   true
    ).

gcd_2_join_loop([],_,_).
gcd_2_join_loop([C # J|Ls],I,N) :-
    (
      C = gcd(M),
      alive(J),
      J \= I ->
        gcd_2_call_body(I,N,M)
    ;   true
    ),
    ( alive(I) ->
        gcd_2_join_loop(Ls,I,N)
    ;   true
    ).

gcd_2_call_body(I,N,M) :-
    ( M =< N ->
        delete(I),
        gcd(M-N)
    ;   true
    ).

gcd_3(I,M) :-
    get_iterator(Ls),
    gcd_3_join_loop(Ls,I,M).

gcd_3_join_loop([],_,_).
gcd_3_join_loop([C # J|Ls],I,M) :-
    (
      C = gcd(N),
      alive(J),
      J \= I ->
        gcd_3_call_body(J,N,M)
    ;   true
    ),
    ( alive(I) ->
        gcd_3_join_loop(Ls,I,M)
    ;   true
    ).

gcd_3_call_body(J,N,M) :-
    ( M =< N ->
        delete(J),
        gcd(M-N)
    ;   true
    ).
```

Figure 2.6: Compiled version of the `gcd` program.

`:-` <*ask-constraint*> `asks` <*tell-constraint*>`.`

The `asks` declaration is effectively a macro definition on which the following restrictions apply. First, each tell constraint can only have one associated ask constraint (although an ask constraint can be associated to more than one tell). Second, the arguments of the *tell-constraint* must be distinct variables, and only these and anonymous variables can appear in the corresponding *ask-constraint*. And finally, the ask constraint must be defined for the type of arguments of the associated tell constraints, it must be usable in any mode in which the associated tell constraints are, and must have `semidet` determinism.

Example 9. *The following are* `asks` *declarations that might be declared by a finite domain solver.*

```
:- 'ask_=<'(X,Y) asks X =< Y.
:- 'ask_=<'(Y,X) asks X >= Y.
```

The CHR compiler uses this information to translate the guards from Example 8 into appropriate ask constraints. □

A predicate is recognised by the compiler as a tell constraint iff it has been declared as having an associated ask constraint. The compiler automatically replaces each such tell constraint which textually appears in a guard with its ask version. In addition, Prolog CHR implementations also allow arbitrary predicates in the guard. This means that tell constraints nested inside the guard will be treated as tells, when perhaps this was not the intention of the programmer. A simple call-graph analysis can detect when this might occur, and hence the compiler can issue a warning.

2.5.2 Guards with Existential Variables

Although normalisation can lead to proliferation of existential variables in the guard, in many cases such existential variables can be compiled away without requiring extra help from the solver. Consider determining whether a constraint store B implies a guard g of the following form

$$v = f(y_1, \ldots, y_n) \wedge g'$$

Where g' represents the rest of the guard, and variable v is existentially quantified. Let $\bar{x}$ be the list of existentially quantified variables excluding v. If f is a total function, such a v always exists and is unique, then as long as none of the variables $y_1, \ldots, y_n$ are existentially quantified (i.e. appear in $\bar{x}$) then the question $\mathcal{D} \models_{\mathcal{S}} B \rightarrow \exists \bar{x} \exists v.g$ is equivalent[3] to the question

$$\mathcal{D} \models_{\mathcal{S}} (B \wedge v = f(y_1, \ldots, y_n)) \rightarrow (\exists \bar{x} g')$$

Now v is no longer existential in g'.

This translation gives the first hint of how to compile guards with existential quantification. Basically the formula $v = f(y_1, \ldots, y_n)$ can be compiled to a tell constraint which constrains a (fresh) variable v to be the result of applying function f to $y_1, \ldots, y_n$, followed by the code for the rest of the guard g'. Note that order is now important, we execute the tell constraint $v = f(y_1, \ldots, y_n)$ before g'.

[3] For incomplete solvers, equivalence is effectively a condition on $(\mathcal{D} \models_{\mathcal{S}})$. In languages where logical connectives (e.g. $\wedge$) are handled by the compiler rather than the built-in solver, we can assume that equivalence holds.

Example 10. *Consider the following CHR constraint* `before_after` *used in task scheduling which extends a finite domain solver. Basically, the constraint* `before_after(`$T1$`,`$D1$`,`$T2$`,`$D2$`)` *holds if the task with start time* $T1$ *and duration* $D1$ *does not overlap with the task with start time* $T2$ *and duration* $D2$*.*

```
before_after(T1,D1,T2,D2) <=> T1 + D1 > T2 | T1 >= T2 + D2.
before_after(T1,D1,T2,D2) <=> T2 + D2 > T1 | T2 >= T1 + D1.
```

Normalisation of the guard results in:

```
before_after(T1,D1,T2,D2) <=> exists [N] (N = T1 + D1, N > T2) |
                                  T1 >= T2 + D2.
before_after(T1,D1,T2,D2) <=> exists [N] (N = T2 + D2, N > T1) |
                                  T2 >= T1 + D1.
```

thus introducing a new existential variable N*.*

The call-body predicate (including the guard test) for the first rule is as follows.

```
before_after_1_call_body(I,T1,D1,T2,D2) :-
    (
      N = T1 + D1,                    % TELL constraint
      'ask_>'(N,T2) ->                % ASK constraint
        delete(I),
        T1 >= T2 + D2                 % Body
    ;   true
    ).
```

Here we assume that the ask version of the $(</2)$ *constraint is* `'ask_<'`$/2$ *with the same arguments. In the first rule the ask constraint* $\exists N(N = T1 + D1)$ *becomes a tell constraint, because neither* `T1` *nor* `D1` *are existentially quantified, and the function* `+` *is total. The same reasoning applies to the second rule.* □

Partial functions are common in Herbrand constraints. Consider the constraint $x = f(y_1, \ldots, y_n)$, where f is a Herbrand constructor. This constraint defines, among others, a partial (deconstruct) function f_i^{-1} from x to each $y_i, 1 \leq i \leq n$. For this reason a compiler can produce a new ask test `bound_f(X)` for each Herbrand constructor f, which check whether X is bound to the function f. Herbrand term deconstructions are then compiled as if the asks declaration

```
:- bound_f(X) asks exists [Y1,..,Yn] X = f(Y1,..,Yn).
```

appeared in the program.

Example 11. *Consider the compilation of the guard from Example 3.*

```
length(Xs1,L) <=> exists [A,Xs] (Xs1 = [A|Xs]) |
                  L = L1+1, length(Xs,L1).
```

We will compile this guard to the call `'bound_[|]'(Xs1)`*.* □

2.6 Summary

In this chapter we have described how to compile a CHR program into a constraint logic programming language. Like most other programming languages, compiling CHRs in a multi-phase process, including phases for parsing and normalisation, guard compilation and code generation, which were all covered here. Also, a reasonably simple runtime environment was described, which implements the execution state for the refined semantics.

Parsing CHR rules is not much different than parsing any other programming language. The only difference is that as CHRs are usually an embedded language, the job of parsing CHRs is left to the parser for the host language. Normalisation is important as it helps simplify later compilation phases. With CHRs we have identified two distinct types of normalisation: guard normalisation and program normalisation. Guard normalisation helps significantly simplify the compilation of the guard later, and program normalisation is especially useful for code generation.

Guard compilation is a surprisingly involved aspect of CHR compilation. This is for several reasons. Firstly, in order to make compilation work for arbitrary built-in solvers a general solver interface was devised, namely mapping tell constraints to ask constraints via a special `asks` declaration. Guards with existentially quantified variables occur often in practice, so a CHR compiler should be able to handle them. Existential variables which are functionally defined by non-existential variables in the guard can be compiled without any additional help from the solver. Unfortunately, a more general solution would require a more complicated interface with the built-in solver.

With the program normalised and the guard compiled, the final phase of CHR compilation is code generation. Many aspects of the CHR execution state map neatly into similar concepts in any constraint logic programming language, e.g. the execution stack becomes the program stack and the built-in store is implemented by the built-in solvers.

The output of the CHR compiler is a series of predicates that manipulate a global CHR constraint store. The CHR compiler replaces each constraint with a so-called *top-level* predicate that performs the necessary overhead before calling the first occurrence. Later, the *occurrence* predicates actually do the rule matching. This mostly involves iterating through the store for matches to individual partners to the active constraint, and this is the role of the *join-loop* predicates. Finally, once potential matches are found, the *call-body* predicate tests the guard and checks the history before actually firing the rule.

The compilation scheme presented in this chapter is very simple, and should give reasonable performance provided the rules themselves have small heads (no more than two heads) and the CHR store never grows too large at runtime. If either of these conditions do not hold then the result is likely to be a very inefficient program. For more complicated programs, an optimizing CHR compiler should be used instead.

Bibliography

B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practices of Constraint Programming*, LNCS 1713, pages 174–188. Springer-Verlag, 1999.

G. Duck, P. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS 3132, pages 90–104. Springer-Verlag, September 2004.

Gregory J. Duck. ToyCHR. http://www.cs.mu.oz.au/∼gjd/toychr/.

T. Frühwirth and P. Brisset. High-level implementations of constraint handling rules. Technical Report ECRC-95-20, ECRC Munich, Germany, 1995.

C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, LNCS 631, pages 260–268. Springer-Verlag, 1992.

Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.

Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, February 1995.

Chapter 3

The K.U.Leuven CHR System

Author:	Tom Schrijvers
Thesis Title:	Analyses, Optimizations and Extensions of Constraint Handling Rules
School:	K.U.Leuven, Belgium
Publication Year:	2005

Foreword

The K.U.Leuven CHR system, together with Duck's system in HAL, formed the first generation of optimizing CHR compilers. Looking at its output, the connection to its predecessor, Holzbaur's CHR compilation schema [Holzbaur and Frühwirth, 1999, 2000] for Prolog, will not be apparent to any but the initiated. The system evolved to incorporate optimizations at many levels and to cater for frequently occurring patterns.

This chapter describes the state of the system and its optimizations about halfway through its development lifetime. Several more optimizations were added later, based on Schrijvers' subsequent research and part of Sneyer's Ph.D. research, notably exploiting optional type and annotations. Currently, the development of the system has ceased; it has become quite stable and requires little further maintenance. Nevertheless, the system is still in quite active use, by both researchers and industrial users, both thanks to its widespread presence in many Prolog systems and its optimizations that make CHR programs with many hundreds of rules effective.

Further optimization developments have taken place in Java CHR system by Van Weert, who has come to show that CHR's optimizing compilation technology is superior to that of other production rules systems. At present, the CHR community is anticipating a new generation of optimized compilation developments.

3.1 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 1.3 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, 2000]. This system was first included in the SICStus 3 release [Intelligent Systems Laboratory, 2003] and later in the Yap Prolog system [Santos Costa et al., 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.

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 et al., 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, 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 et al., 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 et al., 2005]. Section 3.2 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 3.3.

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 3.4.

We verify the realization of the first two objectives with an experimental evaluation of our system. Section 3.5 compares our system to the reference implementation. Finally, Section 3.6 concludes.

3.2 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 et al., 1999] next to the current Mercury [Somogyi et al., 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 [Schrijvers, 2005a, Chapter 5].

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 3.3.

3.3 Optimizations

This section lists the noteworthy optimizations implemented in the K.U.Leuven CHR system. Section 3.3.1 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 3.3.2, Section 3.3.3, and 3.3.4 respectively.

3.3.1 Related Work on Optimizations

Join Ordering In the general compilation schema in [Schrijvers, 2005a, Chapter 5] 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 explained in [Schrijvers, 2005a, Chapter 5] 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 et al., 2005] and we have implemented it in K.U.Leuven CHR system.

Late Storage Schrijvers [Schrijvers, 2005a, Chapter 5] mentions already an optimization to the general compilation schema to postpone constraint storage. In [Holzbaur et al., 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 9.4.

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 et al., 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 et al., 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 et al., 2005b]. This work also contains a prover and optimization to remove redundant guards.

3.3.2 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 deals with ground constraints only.

The implementation of [Holzbaur et al., 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 the 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 9.5 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 [Schrijvers, 2005a, Chapter 5]):

  ```
  ( 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 [Schrijvers, 2005a, Chapter 5]).
- No variables have to be looked for in a ground constraint to associate the constraint suspension with. See also Section 3.3.4 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 9.4.1 and further for a definition of the observation property and an analysis to derive it.

In addition, in Section 3.3.3 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 et al., 2005b,a] 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 et al., 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.

3.3.3 Hash Table Constraint Stores

The CHR constraint store implementation of the general compilation schema, explained in [Schrijvers, 2005a, Chapter 5], 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 et al., 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 [Schrijvers, 2005a, Chapter 4], we have run the CHR program in SWI-Prolog [Wielemaker, 2004] using our system. A discussion of the union-find CHR implementation with rule priorities is given in Example 4.6.3. A version without rule priorities is discussed in Section 6.3.5. For a more detailed discussion we refer the reader to [Schrijvers and Frühwirth, 2005].

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 [Schrijvers, 2005a, Chapter 4] 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 3.1 and 3.2 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 et al., 1990], which is sufficient for our complexity analysis. This is confirmed by experimental evaluation (see Figure 3.3).

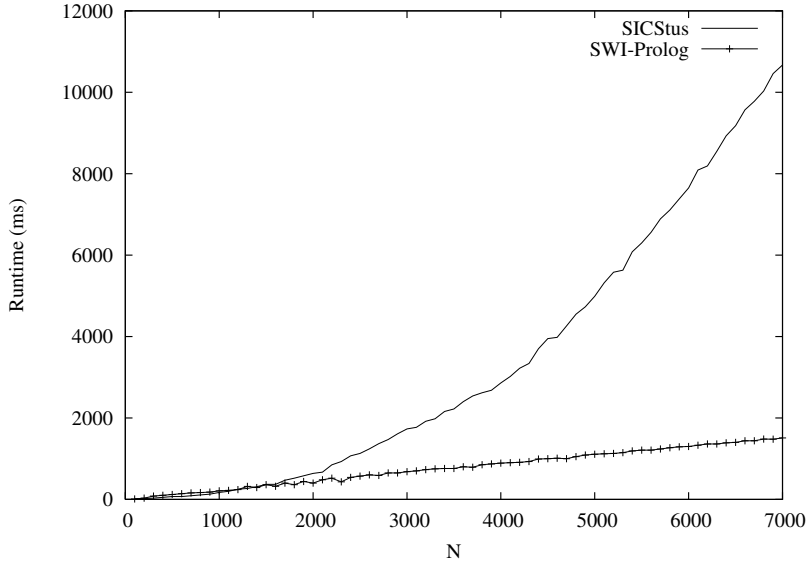


Figure 3.1: Observation of behavior for contrived unions: SICStus and SWI-Prolog array constraint stores

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.

3.3.4 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 anti-monotonic in that guard, if further constraining that argument does not make the guard succeed. Typical examples for Prolog are guards in which the arguments or `var/2` tests do not occur.

As mentioned in [Schrijvers, 2005a, Chatper 5], 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.

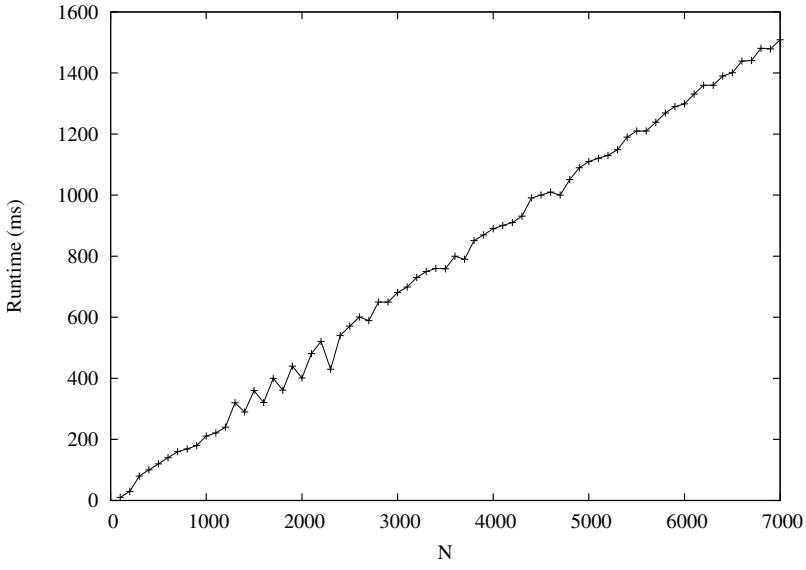


Figure 3.2: Observation of behavior for contrived unions: Detail of Figure 3.1: SWI-Prolog array constraint store

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].

3.4 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 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.

3.4.1 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 [Schrijvers, 2005a, Chapter 8] for our work on integrating the K.U.Leuven CHR system with XSB's

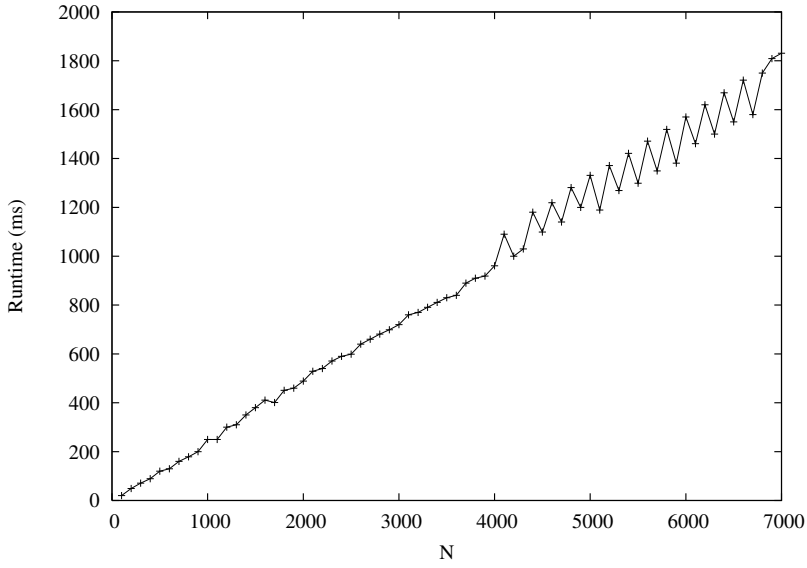


Figure 3.3: Observation of behavior for contrived unions: SWI-Prolog Hash table constraint store

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 [Schrijvers, 2005a, Chapter 5]). This causes unintuitive and unwanted behavior 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 [Schrijvers, 2005a, Chapter 5], was implemented on top of a newly added single global variable that resides at the bottom of XSB's heap.

3.4.2 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 [Schrijvers, 2005a, Chapter 5] 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.

3.5 Experimental Evaluation

3.5.1 Benchmarks

In this Section we evaluate the performance of our CHR system on ten benchmarks which are available from [Schrijvers, 2005b]. 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 3.3.3 with 1,000 elements.

uf_opt Application of the optimal union-find program on the benchmark described in Section 3.3.3 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.

3.5.2 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.

- Secondly, the results may be influenced by the more powerful optimizations of our CHR preprocessor.

	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 3.1: Runtime performance of 8 CHR benchmarks in 5 different Prolog systems.

- 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 3.1 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 3.5.4). The early 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.

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 3.2: Runtime performance of 7 CHR benchmarks optimized with groundness annotations relative to unoptimized programs, in both hProlog and SWI-Prolog.

3.5.3 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 3.2 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.

3.5.4 Anti-monotonic Delay Avoidance

To experimentally evaluate the anti-monotony-based optimization described in Section 3.3.4, we consider the effect on the runtime of our standard set of CHR benchmarks [Schrijvers, 2005b] 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 occurrence of `fibonacci(N,M1)`.

The first rule of `fibonacci1` is:

```
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.
```

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 3.3: Runtime of optimized benchmarks relative to unoptimized ones, in hProlog

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 3.3 lists the runtime results in milliseconds of running the benchmarks in hProlog. The results clearly indicate that there is hardly any effect on the 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%.

3.6 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 et al., 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 et al., 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 et al., 2003].

3.6.1 Future Work

Some steps have already been taken in consolidating the improvements present 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 9. 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.
- 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.

Bibliography

Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java Constraint Kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming, Kiel*, Kiel, Germany, September 2001.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

Bart Demoen. hProlog. http://www.cs.kuleuven.be/ bmd/hProlog/.

Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, October 2002.

Bart Demoen and Phuong-Lan Nguyen. So many WAM variations, so little time. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *CL2000: Proceedings of the 1st International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 1240–1254, Londong, UK, July 2000. ALP, Springer Verlag.

Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An Overview of HAL. In Joxan Jaffar, editor, *CP'99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 174–188, Alexandria, Virginia, USA, 1999. Springer Verlag.

Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In *Artificial Intelligence*, volume 19, pages 17–37, North Holland Conference, 1982.

Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 117–133. Springer Verlag, 1999.

Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.

Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5): 503–531, 2005.

IC-Parc. ECL^{i}PS^e. http://www.icparc.ic.ac.uk/eclipse/.

Intelligent Systems Laboratory. *SICStus Prolog User's Manual*, October 2003.

Micha Meier. Sepia*: The constraint logic programming system. http://www.clps.de/.

Vítor Santos Costa, Luis Damas, Rogerio Reis, and Ruben Azevedo. *YAP User's Manual*, 2004. http://www.ncc.up.pt/~vsc/Yap/.

Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, Department of Computer Science, K.U.Leuven, Belgium, 2005a.

Tom Schrijvers. A Collection of Assorted CHR Benchmarks, 2005b. http://www.cs.kuleuven.be/~toms/Research/CHR/.

Tom Schrijvers and Bart Demoen. Antimonotony-based delay avoidance for CHR. Report CW 385, K.U.Leuven, Department of Computer Science, July 2004a.

Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, Ulm, Germany, May 2004b.

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

Tom Schrijvers and David S. Warren. Constraint handling rules and tabled execution. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 120–136, St-Malo, France, September 2004. Springer Verlag.

Tom Schrijvers, David S. Warren, and Bart Demoen. CHR for XSB. In R. Lopes and M. Ferreira, editors, *CICLOPS 2003: Proceedings of the Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 7–20, Mumbai, India, December 2003. University of Porto.

Tom Schrijvers, Jan Wielemaker, and Bart Demoen. Poster: Constraint Handling Rules for SWI-Prolog. In Armin Wolf, editor, *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, Ulm, Germany, February 2005.

Gert Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer Verlag, Berlin, Germany, 1995.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard Reasoning for CHR Optimization. Report CW 411, K.U.Leuven, Department of Computer Science, Leuven, Belgium, 2005a.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard simplification in CHR programs. In Armin Wolf, editor, *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, pages 123–134, Ulm, Germany, February 2005b.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

Guy Steele. *Common LISP: The Language*. Digital Press, 1984.

Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transations on Programming Languages and Systems*, 2005. To appear.

David S. Warren et al. The XSB Programmer's Manual: version 2.7, vols. 1 and 2, January 2005. http://xsb.sf.net.

Jan Wielemaker. SWI-Prolog release 5.4.0, 2004. http://www.swi-prolog.org/.

Armin Wolf. 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, page 256. Springer Verlag, January 2001.

Armin Wolf. Intelligent Search Strategies Based on Apdative Constraint Handling Rules. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5):567–594, 2005.

Part III

Execution Strategies

Chapter 4

Rule Priorities

Author:	Leslie De Koninck
Thesis Title:	Execution Control for Constraint Handling Rules
School:	K.U.Leuven, Belgium
Publication Year:	2008

Foreword

This chapter introduces CHR^{rp}: Constraint Handling Rules with rule priorities. CHR^{rp} offers flexible control of the propagation strategy which is lacking in CHR. A formal operational semantics for the extended language is given and is shown to be an instance of the theoretical operational semantics of CHR. We relate rule priorities to alternative forms of execution control. We show that CHR^{rp} can be efficiently compiled into the underlying host language. To achieve that, we introduce various optimizations, whose effects are empirically evaluated. The CHR^{rp} compiler is also compared with the state-of-the-art K.U.Leuven CHR system and is shown to exhibit comparable performance while offering a much more high-level form of execution control. This chapter is part of a larger effort to add high-level execution control to CHR. As such it is a fundamental part of the thesis of De Koninck [2008]. It builds further on previous work on operational semantics for CHR, in particular the work defining the refined operational semantics [Duck et al., 2004]. It forms the basis for the proposal of CHR*2* [Van Weert et al., 2009, Van Weert, 2010]. This work sits at the boundary of CHR as a declarative language for implementing constraint solvers, and CHR viewed as a general-purpose programming language. Other work related to execution strategies in CHR is the work on probablistic CHR [Frühwirth et al., 2002] and CHRiSM [Sneyers et al., 2010]. Since this work, it has been shown that CHR with rule priorities is more expressive than regular CHR [Gabbrielli et al., 2009].

4.1 Introduction

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

In this chapter, we propose extending CHR with user-definable rule priorities. The extension, called CHR^{rp}, offers the flexible execution control needed for implementing highly efficient Constraint (Logic) Programming systems. Moreover, it facilitates implementing rule-based algorithms as these often require certain rules to be tried before others, either for efficiency reasons, or for correctness. Rule priorities allow the programmer to write programs that are more concise and efficient, but potentially not confluent under the theoretical operational semantics of CHR. They support a high-level form of execution control that is more declarative, flexible and comprehensible than previously available, while retaining the expressive power needed for the implementation of general purpose algorithms. In CHR^{rp}, all execution control information is explicit in the rule priority annotations, which leads to a clear separation of the logic and control aspects of a CHR^{rp} program in line with [Kowalski, 1979]. We first give some examples of CHR with rule priorities.

Example 4.1.1 (Less-or-Equal)**.** *As a first example, we add priorities to the* `leq` *program of Listing 1.1. The result is shown in Listing 4.1. Note that other priority assignments are also possible.*

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

Listing 4.1: `leq` program in CHR^{rp}

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

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

Example 4.1.2 (Dijkstra's Shortest Path)**.** *A* CHR^{rp} *implementation of Dijkstra's single-source shortest path algorithm is given in Listing 4.2.*

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

Listing 4.2: Dijkstra's shortest path algorithm in CHR^{rp}

The input consists of a set of directed weighted edges, represented as `edge`*/3 constraints where the first and last arguments respectively denote the begin and end nodes, and the middle argument represents the weight. The source node is given by the* `source`*/1 constraint. The algorithm keeps track of upper-bounds on the shortest path distances to the different nodes, represented by* `dist`*/2 constraint whose arguments are respectively the node in question and the distance. Eventually, the distance upper-bounds become tight.*

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

The rest of this chapter is organized as follows. Section 4.2 presents further motivation for extending CHR with rule priorities. In Section 4.3, we formalize the syntax and semantics of the extended language: CHR^{rp}. Two important properties of CHR^{rp} programs, namely confluence and complexity, are discussed in Section 4.4. A basic compilation scheme for CHR^{rp} programs is given in Section 4.5 and several optimizations are proposed in Section 4.6. The resulting CHR^{rp} implementation is empirically evaluated in Section 4.7. In Section 4.8 we discuss related work and Section 4.9 concludes the chapter.

4.2 Motivation and Examples

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

4.2.1 Constraint Propagators

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

Example 4.2.1. *We represent a binary constraint c between two variables x and y as* $\texttt{c}(c, x, y)$*. A CHR constraint* $\texttt{d}(x, dx)$ *represents that variable x has domain dx. An implementation of two types of constraint propagators for such constraints is given in Listing 4.3, where the Prolog built-in predicate* `member(X,L)` *non-deterministically unifies* `X` *with one of the values of the list* `L` *and* `findall/3` *is used to retrieve a list (3rd argument) of all the objects (1st argument) that satisfy a given goal (2nd argument).*

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

At first glance it might appear that, given the textual order of the rules, the refined operational semantics of CHR ensures that the arc consistency rules are always tried before the path consistency rules. The following situation shows that this is not always the case.

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

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

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

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

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

Listing 4.3: Arc and path consistency propagators

Let x be a variable whose domain dx_0 has changed to dx_1 by using one of the arc consistency rules with as active constraint the domain of some variable y. Assume that x is arc consistent. The constraint $\mathtt{d}(x, dx_1)$ *becomes active and the rules* $\mathtt{ac_1}$ *and* $\mathtt{ac_2}$ *are tried, none of which fires. At that moment, rule* $\mathtt{pc_1}$ *will be tried, while there might still be a variable (e.g. y) that is not arc consistent yet. In contrast, rule priorities ensure that these variables are made arc consistent first.*

4.2.2 Soft Constraints

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

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

Example 4.2.2. *We consider variables over finite integer domains, again represented as* $\mathtt{d}(x, dx)$ *constraints, with x a variable and dx a domain, which is a list of pairs $v - s$ with v a value (an integer) and s a score (a floating point number between 0 and 1). The solver of Listing 4.4 implements three constraints, as well as a simple labeling procedure. The first constraint,* `odd/1`*, is a hard constraint stating that the variable that is its argument, can only take an odd value. The second constraint* $\mathtt{lt}(x, y, p)$ *is a soft constraint that imposes a penalty of p to combinations of values for x and y for which x is not less than y. Finally, there is the* $\mathtt{all\_different}(x, y, z, p)$ *constraint, which is again a soft constraint and which imposes a penalty of p to combinations of values for x, y and z in*

which these variables do not all have different values. The soft constraints are dealt with

```
% hard constraint

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

% soft constraint over two variables

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

% soft constraint over three variables

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

% labeling

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

Listing 4.4: Solver for hard and soft constraints

by first combining the values of the variables involved, and then projecting the combinations on the different component variables using `project_on_fst`*/2,* `project_on_snd`*/2 and* `project_on_trd`*/2. The code for the projection predicates is given in Listing 4.5. It uses the Prolog predicate* `select/3`*, which removes a given value from the first list, and returns the resulting results in the third argument. Basically, the program combines the scores for all tuples with the same value for respectively their first, second or third component, using the maximum operation. The* `all_different`*/4 soft constraint is more expensive to enforce than the* `lt`*/3 constraint, and so it is given a lower priority. The hard*

constraint odd/1 *is given the highest priority, and the labeling rule the lowest. Note the guard in the labeling rule that prevents non-terminating behavior resulting from replacing* d/2 *constraints by identical versions.*

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

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

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

Listing 4.5: Projection for soft constraints

4.2.3 Constraint Store Invariants

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

Example 4.2.3. *Consider that we want to check whether two graphs, G_1 and G_2 are equal. We do this by removing those edges that are common to both graphs. If there are still edges after reaching a fixed point, then the graphs are different. We represent the edges of graph G_1 and G_2 by the edge constraints* $\mathtt{e}_1/2$ *and* $\mathtt{e}_2/2$ *respectively. This gives us the program of Listing 4.6.*

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

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

Listing 4.6: Constraint store invariants in CHR$^{\mathrm{rp}}$

Edges obey set semantics, as implemented by the rules $\mathtt{s}_1$ *and* $\mathtt{s}_2$*. The rule* rc *(**r**emove **c**ommon) removes those edges that appear both in graph G_1 and in graph G_2. Now consider the goal $G = \{\mathtt{e}_1(\mathtt{X},\mathtt{X}), \mathtt{e}_2(\mathtt{X},\mathtt{Y}), \mathtt{e}_2(\mathtt{Y},\mathtt{X}), \mathtt{X} = \mathtt{Y}\}$. Under the refined operational semantics, ignoring the rule priorities, this goal is executed from left to right. Solving the built-in constraint* $\mathtt{X} = \mathtt{Y}$ *causes the sequential activation of all three CHR constraints. If the* $\mathtt{e}_1/2$ *constraint is activated before any of the* $\mathtt{e}_2/2$ *constraints, then rule* rc *fires before the* $\mathtt{s}_2$ *rule is tried, which results in a final constraint store containing the* $\mathtt{e}_2(\mathtt{X},\mathtt{X})$ *constraint. This erroneously indicates that the graphs are different.*

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

The above example can *be implemented correctly using the refined semantics, but this leads to inefficient and unreadable code as shown in Listing 4.7.*

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

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

Listing 4.7: Constraint store invariants in regular CHR

4.2.4 Dynamic Rule Priorities

Rule priorities are called *dynamic* if they depend on (the arguments of) the constraints that form a rule instance. Dynamic rule priorities are only known at runtime and different instances of the same rule may have a different priority. In Example 4.1.2 we have already shown how to implement Dijkstra's shortest path algorithm using dynamic rule priorities. Below, another example program is given.

Example 4.2.4 (Sudoku)**.** *The Sudoku solver from Section 1.5.1 keeps track of the number of possible values for each Sudoku cell and chooses a value from the most constrained cell first. Listing 4.8 shows the labeling code.*

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

Listing 4.8: Labeling rules for a Sudoku solver in CHR

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

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

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

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

Table 4.1: Transitions of ω_p

The remaining rules of the program filter out inconsistent values:

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

4.3 CHR^{rp} CHR with Rule Priorities

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

4.3.1 Syntax

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

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

where r, H^k, H^r, G and B are as defined in Section 1.1.2. The rule *priority* p is an arithmetic expression for which holds that $\mathsf{vars}(p) \subseteq (\mathsf{vars}(H^k) \cup \mathsf{vars}(H^r))$, i.e., all variables in p also appear in the heads. A rule in which $\mathsf{vars}(p) = \emptyset$ is called a *static* priority rule: its priority is known at compile time and equal for all rule instances. A rule in which $\mathsf{vars}(p) \neq \emptyset$ is called a *dynamic* priority rule: its priority is only known at runtime and different instances of the same rule may fire at different priorities. In this chapter, all rule priorities are integers or arithmetic expressions that evaluate on integers. In general, rule priorities could be any type of terms for which we have a total preorder.

4.3.2 Operational Semantics

We propose a formal operational semantics for CHR^{rp}. It is called the priority semantics and denoted by ω_p. It consists of a refinement of the ω_t semantics of CHR with a minimal amount of determinism in order to support rule priorities. The ω_p semantics uses the same state representation as the ω_t semantics. Its transitions are shown in Table 4.1. The ω_p semantics restricts the applicability of the **Apply** transition with respect to the ω_t semantics. It is only applicable to states with an empty goal and it fires a rule instance of

priority p in state σ only if there exists no ω_t **Apply** transition $\sigma \overset{\omega_t}{\rightarrowtail}_P \sigma'$ that fires a rule instance of a higher priority. The **Solve** and **Introduce** transitions are unchanged.

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

Example 4.3.1. *Consider the following program:*

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

Using the refined operational semantics, the rule priority declarations are ignored. For the initial goal $G = \{\mathtt{a}\}$, we get the following results. Under ω_p, the unique qualified answer for goal G is $\{\mathtt{b}, \mathtt{c}\}$ whereas under the ω_r semantics, the answer is $\{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The difference is explained as follows: under the ω_p semantics, constraint $\mathtt{a}$ is removed by rule $\mathtt{r_3}$ before rule $\mathtt{r_4}$ is tried. This causes rule $\mathtt{r_4}$ to be no longer applicable. There is no rule ordering that can cause rule $\mathtt{r_3}$ to be fired after rule $\mathtt{r_2}$ but before rule $\mathtt{r_4}$ in the ω_r semantics.

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

Example 4.3.2. *Consider the following program:*

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

In this example, rules $\mathtt{r_1}$ and $\mathtt{r_2}$ have an equal priority. For the initial goal $\mathtt{a}$, we get the following output:

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

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

Example 4.3.3. *Consider the following program:*

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

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

Output in ω_r	*Alternative Outputs in ω_p*			
$\mathtt{r_1} : 1$	$\mathtt{r_1} : 1$	$\mathtt{r_1} : 1$	$\mathtt{r_1} : 2$	$\mathtt{r_1} : 2$
$\mathtt{r_2} : 1$	$\mathtt{r_1} : 2$	$\mathtt{r_1} : 2$	$\mathtt{r_1} : 1$	$\mathtt{r_1} : 1$
$\mathtt{r_1} : 2$	$\mathtt{r_2} : 1$	$\mathtt{r_2} : 2$	$\mathtt{r_2} : 1$	$\mathtt{r_2} : 2$
$\mathtt{r_2} : 2$	$\mathtt{r_2} : 2$	$\mathtt{r_2} : 1$	$\mathtt{r_2} : 2$	$\mathtt{r_2} : 1$

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

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

Example 4.3.4. *A form of negation by absence (see also [Van Weert et al., 2006]) can be implemented in* CHR^{rp} *as follows:*

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

Under the ω_t semantics, the goal {a, no_a} *either fails or succeeds with qualified answer* {a}*, whereas under the ω_p semantics (as well as the refined semantics) it must fail and in particular, the second rule cannot fire. The goal* {no_a} *succeeds under both semantics by means of firing rule* r_2.

4.3.3 Correspondence between ω_p and ω_t Derivations

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

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

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

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

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

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

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

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

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

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

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

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

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

Theorem 3 implies that for CHR$^{\mathrm{rp}}$ programs in which all rule priorities are equal, every execution strategy under ω_t is consistent with ω_p, and so such programs can be executed using the refined operational semantics as implemented by current CHR implementations. While such CHR$^{\mathrm{rp}}$ programs are obviously degenerate, we can retain many advantageous aspects of the refined operational semantics of CHR when compiling CHR$^{\mathrm{rp}}$ programs (see Section 4.5).

4.4 Program Properties

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

4.4.1 Confluence

Confluence is the property that a program always gives the same answers for a given goal, regardless of the execution path followed. In [Abdennadher, 1997], it is shown that for terminating programs, confluence under the theoretical operational semantics of CHR can

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

be decided by checking if all *critical pair* states are joinable. A critical pair consists of two *minimal* states σ_1 and σ_2 that result from a common ancestor state σ after firing different rule instances, such that the rule instance that led to σ_1 cannot fire in σ_2 and vice versa. Two states are joinable if they both derive into states that are *variants* of each other. More formal definitions can be found in [Abdennadher, 1997]. Clearly, under the ω_p semantics, a critical pair only follows from firing rule instances with equal priority.

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

Example 4.4.1. Consider the following example, adapted from [Duck, 2005] and extended with rule priorities:

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

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

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

with common ancestor state

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

Both states further derive into $\langle\emptyset, \emptyset, \mathtt{true}, T'\rangle_{n+1}$ and the program appears to be confluent. However, given the initial goal $\{\mathtt{p}, \mathtt{q(1)}, \mathtt{q(2)}, \mathtt{q(3)}\}$ we get the qualified answers $\{\mathtt{q(1)}, \mathtt{q(2)}\}$, $\{\mathtt{q(2)}, \mathtt{q(3)}\}$ or $\{\mathtt{q(1)}, \mathtt{q(3)}\}$. The problem is that in a minimal state, rule $\mathtt{r_2}$ is not applicable and rules $\mathtt{r_3}$ and $\mathtt{r_4}$ can fire. In a larger state, $\mathtt{r_2}$ may become applicable and cause rules $\mathtt{r_3}$ and $\mathtt{r_4}$ to be not applicable anymore. This example shows that the fact "all critical pairs are joinable" does not necessarily imply local confluence (and hence confluence) under $\mathrm{CHR}^{\mathrm{rp}}$.

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

Corollary 4.4.2. *If program P is terminating under ω_p, and confluent under ω_t, then it is also confluent under ω_p.*

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

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

Example 4.4.3. *Consider the* `leq` *program from Example 1.1.1. This program is confluent under ω_t. Therefore, by Corollary 4.4.2, the* `leq` *program is confluent under ω_p.*

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

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

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

Practical Confluence Test

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4.2 Complexity

In Chapter 7, a meta-complexity result for CHR^rp is given that allows one to derive the time complexity of a CHR^rp program by reasoning amongst others about the number of rule applications. The approach is based on the Logical Algorithms formalism by Ganzinger and McAllester [2002] and relies on an optimized implementation of CHR^rp. The result of Sneyers et al. 2008 that states that any algorithm can be implemented in CHR with optimal time and space complexity, also easily transfers to the CHR^rp context. However, it requires memory optimizations that are currently not implemented by our CHR^rp compiler (see [Sneyers et al., 2006b]).

4.5 Basic Compilation of CHR^rp

This section gives an overview of the basic compilation schema for CHR^rp programs. First, in Section 4.5.1, we present a refinement of the ω_p semantics that follows the actual implementation more closely. This refinement, called the refined priority semantics and denoted by ω_{rp}, is based on the refined operational semantics ω_r of (regular) CHR and is as such also based on lazy matching and the concept of active constraints. The ω_{rp} semantics requires that each active constraint determines the actual (ground) priorities of all rules in which it may participate. In Section 4.5.2, we show how dynamic priority rules can be transformed so that this property holds for all active constraints. Finally, Section 4.5.3 gives an abstract version of the code generated for each of the ω_{rp} transitions.

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

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

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

Table 4.2: Transitions of ω_{rp}

4.5.1 The Refined Priority Semantics ω_{rp}

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

The transitions of the ω_{rp} semantics are shown in Table 4.2. The main differences compared to the ω_r semantics are the following. Instead of adding new or reactivated constraints to the activation stack, the **Solve** and **Schedule**[5] transitions schedule them for activation, once for each priority at which they have occurrences. The **Activate** transition activates the highest priority scheduled constraint if it has a higher priority than the current active constraint (if any). This transition only applies if the **Solve** and **Schedule** transitions are not applicable, i.e., after processing the initial goal or a rule body. Its function is similar to that of the ω_r **Reactivate** transition, except that it also

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

applies to constraints that have never been activated before. Noteworthy is that once a constraint is active at a given priority, it remains so at least until a rule fires or it is made passive by the **Drop** transition. Hence we should only check the priority queue for a higher priority scheduled constraint at these program points. Again, the transitions are exhaustively applied starting from an initial state $\langle G, \emptyset, \emptyset, \texttt{true}, \emptyset\rangle_1$ with G the goal, given as a sequence.

Example 4.5.1. *Consider the* `leq` *program of Listing 4.1 and the goal*

$$\{\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(B,A)}\}.$$

The ω_{rp} state after three initial ***Schedule*** *transitions is as follows, where we write the shorthand $c\#i$ @ $\{p_1, \ldots, p_n\}$ for $\{c\#i$ @ $p_1, \ldots, c\#i$ @ $p_n\}$:*

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

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

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

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

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

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

This last transition reschedules the `leq(B,C)#2` *constraint at priorities 1 and 2. None of the remaining constraints in the schedule lead to a rule firing.*

4.5.2 Transforming Dynamic Priority Rules

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

Example 4.5.2. *Consider the rule*

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

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

In this section, we present a pseudo-code source-to-source transformation to transform a program such that the required property is satisfied. In what follows, we refer to the *join order* for a given constraint occurrence, which is the order in which the partner constraints for this occurrence are retrieved (by nested loops). We consider a join order Θ to be a

permutation of $\{1,\ldots,n\}$ where n is the number of heads of the rule. Now, consider a dynamic priority rule

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

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

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

where *Vars* are the variables shared by the first k heads on the one hand, and the remaining heads, the guard, the body and the priority expression on the other, i.e., $\mathit{Vars} = \big(\cup_{i=1}^{k}\mathsf{vars}(C_{\Theta(i)})\big) \cap \big(\big(\cup_{i=k+1}^{n}\mathsf{vars}(C_{\Theta(i)})\big) \cup \mathsf{vars}(g\wedge B\wedge p)\big)$. The first rule generates a partial match that knows its priority once the necessary arguments are ground (fixed). It runs at the highest possible value of the dynamic priority expression.[6] The second rule ensures that the priority expression is ground before the partial match is scheduled at its dynamic priority. The rule runs at the same priority as the first one. Finally, the third rule extends the partial match (with ground priority) into a full match. There we check whether all constraints in the partial match are still alive (calls to `alive`/1), and delete the removed heads (calls to `kill`/1). The *pragma*[7] `passive`/1 denotes that a given head is passive, i.e., no occurrence code is generated for it (see further in Section 4.5.3). The pragma `history`/1 states the tuple layout for the propagation history. All rule copies share the same history which ensures that each instance of the original rule can fire only once.

Example 4.5.3. *Given the rule* `r` *of Example 4.5.2 and join orders* $\Theta_1=[1,2,3]$, $\Theta_2=[2,3,1]$ *and* $\Theta_3=[3,2,1]$.[8] *Furthermore assuming we schedule at a dynamic priority as soon as we know it, we generate the rules of Listing 4.9.*

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

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

4.5.3 Compilation Schema

Now that we have shown how a program can be transformed such that each constraint knows the priorities of all rules in which it may participate, we are ready to present the

[6] We assume 1 is an upper-bound. A tighter one can be used instead if such is known.
[7] Most CHR systems support compiler directives by using the keyword `pragma`.
[8] By slight abuse of notation, we denote $\Theta(1)=\theta_1,\ldots,\Theta(n)=\theta_n$ by $\Theta=[\theta_1,\ldots,\theta_n]$.

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

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

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

Listing 4.9: Example output of the dynamic priority rule transformation

compilation schema. The generated code follows the ω_{rp} semantics closely. In what follows, we assume the host language is Prolog, although the compilation process easily translates to other host languages as well. We note that the generated code presented in this section, closely resembles that of regular CHR under the refined operational semantics, as described in for example [Schrijvers, 2005]. The differences correspond to those between ω_r and ω_{rp} as given in Section 4.5.1.

CHR Constraints

Whenever a new CHR constraint is asserted, it is scheduled at all priorities at which it may fire (**Schedule** transition). Furthermore, it is attached to its variables for the purpose of facilitating the **Solve** transition. In Prolog this is done using attributed variables. The idea is similar to that of subscribing to event notifiers. Finally, the constraint is inserted into all indexes on its arguments. Schematically, the generated code is as shown in Listing 4.10.

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

Listing 4.10: Generated code for a new CHR constraint

The *GenerateSuspension* code creates a data structure (called the *suspension term* in CHR terminology) for representing the constraint in the constraint store. It has amongst others fields for the constraint identifier, its state (dead or alive), its propagation history,[9]

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

its arguments, and pointers for index management. The scheduling code consists of insertions of calls to the code for the first occurrence of each priority p_i ($1 \leq i \leq m$) into the priority queue. With respect to the usual code for CHR constraints under the ω_r semantics, we have added the `schedule`/2 calls and removed the call to the code for the first occurrence of the constraint.

Built-in Constraints

Built-in constraints are dealt with by the underlying constraint solver, in this case the Prolog Herbrand solver. Whenever this solver binds a variable to another variable or a term (during unification), a *unification hook* is called (see [Holzbaur, 1992, Demoen, 2002]). In this hook, the CHR part of the **Solve** transition is implemented. It consists of reattaching the affected constraints, updating the indexes, and scheduling the affected constraints again at each priority for which they have occurrences.

Occurrence Code

For each constraint occurrence, a separate predicate is generated, implementing the **Simplify** and **Propagate** transitions. Its clauses are shown schematically in Listing 4.11. The approach is very similar to how occurrences are compiled under the refined operational semantics of CHR. The differences are that only the occurrences of the same priority are linked, where occurrences with a dynamic priority are assumed to run at different priorities, and the priority queue is checked (`check_activation`/1) after each rule application. The code shown is for the j^{th} priority p occurrence of the c/n constraint which is in an m-headed rule. The indices $r(1), \ldots, r(i)$ refer to the removed heads.

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

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

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

4.6.1 Reducing Priority Queue Operations

A first optimization consists of only scheduling the highest priority occurrence of every new constraint. Only when the constraint has been activated at this priority and has gone through all of its occurrences without being deleted, it is scheduled for the next priority. This is a simple extension of how we linked occurrences of equal priority in the occurrence code.

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

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

...

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

Listing 4.11: Generated occurrence code

In the basic compilation scheme, it is checked whether a higher priority scheduled constraint exists after each rule firing. In a number of cases, this is not needed. If the active constraint is removed, it is popped from the top of the activation stack and the activation check that caused it to be activated, checks again to see if other constraints are ready for activation. So, since a priority queue check will take place anyway, we do not need to do it here. If the body of a rule does not contain CHR constraints with a priority higher than the current one, nor built-in constraints that can trigger any CHR constraints to be scheduled at a higher priority, then after processing the rule body, the active constraint remains active and we do not need to check the priority queue. We call the above optimizations *reduced activation checking*.

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

Example 4.6.1. *We illustrate the applicability of the proposed optimizations on the* `leq` *program given in Listing 4.1. The* `leq/2` *constraint has five occurrences at priority 1 and two at priority 2. New* `leq/2` *constraints are only scheduled at priority 1. Only if an activated constraint has passed the last priority 1 occurrence, it is scheduled at priority 2. For the first three priority 1 occurrences, as well as for the removed occurrence in the* `idempotence` *rule, the active constraint is removed and so there is no need to check the priority queue after processing the rule body. Since the body of the remaining priority 1 occurrence equals* `true`*, no higher priority constraint is scheduled and so we do not need to check the queue here either. Finally, for the* `transitivity` *rule we have that the only constraint in the body has a higher priority occurrence than the current active occurrence, and so we can apply inline activation there.*

4.6.2 Late Indexing

Similar to an optimization from regular CHR, we can often postpone storage of constraints, reducing cost if the constraint is removed before the storage operations are to be applied. We extend the late storage concept of [Holzbaur et al., 2005] to late indexing, where we split up the task of storing a constraint into the subtasks of inserting it into different indexes. The main idea is that an active constraint can only be suspended by another constraint for occurrences of that constraint in rules of a higher priority. This implies that when a constraint is active at a given current priority, it should only be stored in those indexes that are used by higher priority rules.

Example 4.6.2. *In the* `leq` *program (Listing 4.1), the* `leq/2` *constraints are indexed*

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

By using late indexing, new `leq/2` *constraints are not indexed at the moment they are asserted, but only scheduled (and this only at priority 1). When an active* `leq/2` *constraint*

'survives' the last priority 1 occurrence, it is indexed on the combination of both arguments and rescheduled at priority 2. We can postpone the indexing this long because only one constraint can be on the execution stack for each priority and hence all partner constraints have either been indexed already, or still need to be activated. Only after a reactivated `leq/2` *constraint has passed the last priority 2 occurrence, it is stored in the remaining indexes. Note that our approach potentially changes the execution order of the program, which can sometimes contribute to changes in the running time (in either direction).*

4.6.3 Passive Occurrences

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

Example 4.6.3 (Naive Union-Find)**.** *Listing 4.12 shows a naive* CHR^{rp} *implementation of the union-find algorithm (see e.g. [Tarjan and van Leeuwen, 1984]) and is adapted from [Schrijvers and Frühwirth, 2006].*[10]

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

Listing 4.12: Naive union-find in CHR^{rp}

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

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

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

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

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

4.7 Benchmark Evaluation

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

Less-or-Equal The `leq` benchmark uses the program of Listing 4.1 and for given n, the initial goal $G = G_1 \cup G_2$ with

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

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

Because of the *batch* semantics of CHR$^{\mathrm{rp}}$ (i.e., the constraints from the goal are all inserted into the store before the first of them is activated), we achieve an almost linear time complexity (in n) for this benchmark because of the order in which constraints are activated.[11] Noteworthy is that by using the late indexing optimization, we get a higher complexity because the necessary partner constraints for the optimal firing order are not yet stored. This is illustrated in Figure 4.1, which shows runtimes for the `leq` benchmarks in three different setups. In particular, we have tested our CHR$^{\mathrm{rp}}$ system with late indexing (Priority, +LI) and without late indexing (Priority, -LI), and have also compared with regular CHR under the ω_r semantics (Refined) using the K.U.Leuven CHR system (with all optimizations turned on). The benchmarks were run on a Pentium IV, 2.8GHz running SWI-Prolog version 5.6.55. The results do not include garbage collection times. Even with late indexing, our implementation performs better than the K.U.Leuven CHR system for large enough values of n. This is mainly due to better indexing used in our system.

Optimizations Table 4.3 shows benchmark results for various programs where the effect on the runtime of each of the optimizations is measured. The runtimes are given as percentages of the runtime of the unoptimized version for each program. For the unoptimized and fully optimized versions, we also give times in seconds. We use the same setup as in the previous paragraph. The `loop` benchmark consists of the following two rules (and does not rely on priorities):

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

and initial goal $\{\mathtt{a}(2^{20})\}$. The `leq` benchmark is based on the one presented in the previous paragraph. However, to measure the effects of late indexing properly, we ensure that the same rule instances fire in both the versions with and without late indexing. We do so by first asserting the subgoal G_1, waiting for a fixpoint, and only then asserting

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

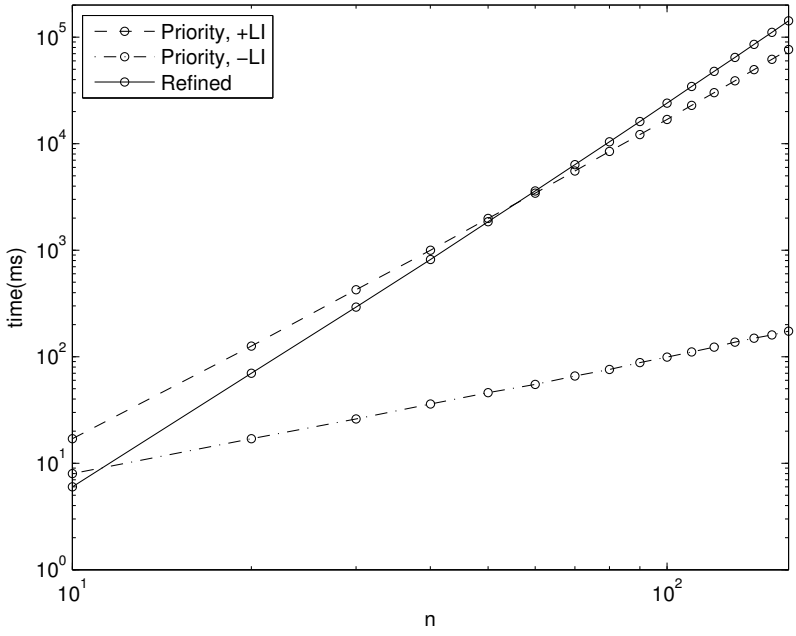


Figure 4.1: Benchmark results for `leq`

subgoal G_2. We have measured for $n = 80$. The `dijkstra` benchmark uses the program of Example 4.1.2 with a graph of 2^{15} nodes and $3 \cdot 2^{15}$ edges; the `union-find` benchmark uses the program of Example 4.6.3 with 2^{12} random `union`/2 constraints over an equal number of elements; and finally the `sudoku` benchmark uses the program of Example 4.2.4 and solves a puzzle in which initially 16 cells have a value (Figure 4.2).[12]

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

Figure 4.2: Benchmark Sudoku puzzle

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

The inline activation analysis assumes that dynamic priority rules run at the highest possible value of the priority expression. It currently assumes this value is 1, but a bounds analysis or a user declaration can give a tighter upper-bound. In the `dijkstra` and `sudoku`

[12]This puzzle can be solved without backtracking, but this requires a stronger form of consistency, and relies on the observation that a symmetric solution can be found by switching numbers 8 and 9.

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

Table 4.3: Benchmark results

benchmarks, we have used a tight upper-bound of 2 for the dynamic priority rules. The passive analysis applied to the `union-find` benchmark cuts off another 1% and reduces the runtime with full optimization to about 16% of the runtime without optimization. In the `loop` benchmark, inline activation only has a strong effect in combination with reduced activation checking: the combined optimizations reduce the runtime by 42% whereas the individual optimizations only cause a reduction of respectively 11% and 2%. The late indexing optimization can change the execution order. We have already shown how this affects the `leq` benchmark. Similarly, it also affects the `sudoku` benchmark which has (amongst others) 11% more rule applications in the version *with* late indexing, hence the increase in runtime. Moreover, late indexing only reduces the amount of index insertions by 3% in this benchmark. Therefore, in this case we get the best result, namely a runtime of 15.68 seconds, when all optimizations except for late indexing are turned on.

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

For the `dijkstra` benchmark, we compared with the CHR program given in [Sneyers et al., 2006a].[13] Our implementation runs about 2.4 times slower than the (regular) CHR implementation, but it is also arguably more high-level. Noteworthy is the following optimization, implemented in [Sneyers et al., 2006a] and reformulated here in terms of our CHR$^{\text{rp}}$ implementation. The rule

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

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

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

15% slower.[14]

4.8 Related Work

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

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

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

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

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

The ECL^{i}PS^e Constraint Logic Programming system [Wallace et al., 1997] supports execution at 12 different priorities. A goal G is executed at a given priority p by using `call_priority`(G, p). The priority system is shared by all constraint libraries, which allows for a form of global control over cooperating constraint solvers. The Extended Constraint Handling Rules library (`ech`) of ECL^{i}PS^e uses the priority system to support a form of static constraint priorities. See [De Koninck, 2008] for more details.

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

4.9 Conclusion

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

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

Bibliography

Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 1997.

Jos C.M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Term rewriting systems with priorities. In Pierre Lescanne, editor, *2nd International Conference on Rewriting Techniques and Applications*, volume 256 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 1987.

Stefano Bistarelli, Thom Frühwirth, Michael Marte, and Francesca Rossi. Soft constraint propagation and solving in Constraint Handling Rules. *Computational Intelligence: Special Issue on Preferences in AI and CP*, 20(2):287–307, 2004.

Gerhard Brewka and Thomas Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.

James C. Browne, E. Allen Emerson, Mohamed G. Gouda, Daniel P. Miranker, Aloysius K. Mok, Roberto J. Bayardo, Jr, Sarah E. Chodrow, David Gadbois, F. Furman Haddix, Thomas W. Hetherington, Lance Obermeyer, Duu-Chung Tsou, Chic-Kan Wang, and Rwo-Hsi Wang. A new approach to modularity in rule-based programming. In *6th International Conference on Tools with Artificial Intelligence*, pages 18–25. IEEE Computer Society, 1994.

Ricardo Caferra, Rachid Echahed, and Nicolas Peltier. Rewriting term-graphs with priority. In Annalisa Bossi and Michael J. Maher, editors, *8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 109–120. ACM, 2006.

Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.

Leslie De Koninck. *Execution control for Constraint Handling Rules.* PhD thesis, K.U.Leuven, 2008.

Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Technical Report CW 350, Department of Computer Science, K.U.Leuven, 2002.

Gregory J. Duck. *Compilation of Constraint Handling Rules.* PhD thesis, University of Melbourne, 2005.

Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.

Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

Ernest Friedman-Hill. *JESS 7.0p2: The rule engine for the Java platform*, 2007. http://herzberg.ca.sandia.gov/jess.

Thom Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic Constraint Handling Rules. *Electronic Notes in Theoretical Computer Science*, 76:115–130, 2002.

Maurizio Gabbrielli, Jacopo Mauro, and Maria Chiara Meo. On the expressive power of priorities in CHR. In *11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 267–276. ACM, 2009.

Harald Ganzinger and David A. McAllester. Logical algorithms. In Peter J. Stuckey, editor, *18th International Conference on Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2002.

Alejandro Javier García and Guillermo Ricardo Simari. Defeasible logic programming: An argumentative approach. *Theory and Practice of Logic Programming*, 4(1-2):95–138, 2004.

Joseph C. Giarratano. *CLIPS User's Guide, Version 6.20*, 2002. http://www.ghg.net/clips/CLIPS.html.

Christian Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In Maurice Bruynooghe and Martin Wirsing, editors, *4th International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 260–268. Springer, 1992.

Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(4 & 5):503–531, 2005.

ILOG CPLEX 7.5 reference manual. ILOG, 2001a.

ILOG Solver 5.1: Reference Manual. ILOG, 2001b.

Robert A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7): 424–436, 1979.

Daniel P. Miranker. TREAT: A better matching algorithm for AI production system matching. In *6th National Conference on Artificial Intelligence*, pages 42–47. AAAI Press, 1987.

Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *8th National Conference on Artificial Intelligence*, pages 685–692. AAAI Press / The MIT Press, 1990.

Mark Proctor, Michael Neale, Michael Frandsen, Sam Griffith, Jr, Edson Tirelli, Fernando Meyer, and Kris Verlaenen. *Drools Documentation, Version 4.0.3*, 2007. http://www.jboss.com/products/rules.

Georg Ringwelski and Matthias Hoche. Impact- and cost-oriented propagator scheduling for faster constraint propagation. In Armin Wolf, Thom Frühwirth, and Marc Meister, editors, *19th Workshop on (Constraint) Logic Programming*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 88–98. Universität Ulm, 2005.

Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, 2005.

Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *1st Workshop on Constraint Handling Rules: Selected Contributions*, volume 2004-01 of *Ulmer Informatik-Berichte*, pages 1–5. Universität Ulm, 2004.

Tom Schrijvers and Thom Frühwirth. Optimal union-find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.

Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633. Springer, 2004.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In Michael Fink, Hans Tompits, and Stefan Woltran, editors, *20th Workshop on Logic Programming*, INFSYS Research Report 1843-06-02, pages 182–191. TU Wien, 2006a.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. Memory reuse for CHR. In Sandro Etalle and Miroslaw Truszczynski, editors, *22nd International Conference on Logic Programming*, volume 4079 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2006b.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. To appear in ACM Transactions on Programming Languages and Systems, 2008.

Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. *Theory and Practice of Logic Programming*, 10(4-6):433–447, 2010.

Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

Peter Van Weert. *Extension and optimising compilation of Constraint Handling Rules*. PhD thesis, K.U.Leuven, 2010.

Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In Tom Schrijvers and Thom Frühwirth, editors, *3rd Workshop on Constraint Handling Rules*, Report CW 452, pages 125–140. Department of Computer Science, K.U.Leuven, 2006.

Peter Van Weert, Leslie De Koninck, and Jon Sneyers. A proposal for a next generation of CHR. In *6th Workshop on Constraint Handling Rules*, pages 1–17, 2009.

Mark Wallace, Stefano Novello, and Joachim Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.

Jennifer Widom. The Starburst rule system. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, pages 87–109. Morgan Kaufmann, 1996.

Chapter 5

Concurrent CHR

Author:	Edmund S.L. Lam
Thesis Title:	Parallel Execution of Constraint Handling Rules – Theory, Implementation And Application
School:	National University of Singapore, Singapore
Publication Year:	2010

Foreword

The abstract CHR semantics essentially involves multi-set rewriting over a multi-set of constraints. This computational model is highly concurrent as theoretically rewriting steps over non-overlapping multi-sets of constraints can execute concurrently [Frühwirth, 2009]. Most intriguingly, this introduces the possibility for a highly parallel CHR solver implementation, which can be used as a high-level general purpose parallel programming language. This means that we can naturally use CHR as a high-level concurrency abstraction which allows us to focus on programming the synchronization of concurrent resources and processes, rather than on micro-managing the concurrent accesses of shared memory.

In the coming sections, we detail our view of concurrency in CHR and the idea of CHR as a parallel programming language. We introduce a concurrent goal-based execution model for CHR. These execution model differentiates itself from existing ones, in that it explicitly describes concurrent derivation steps initiated by multiple active CHR goals. Following this, we introduce a parallel implementation of CHR in Haskell, based on this concurrent goal-based execution model. We will detail the issues which must be addressed in order to achieve a practical implementation that scales well with system resources. Finally, we demonstrate the scalability of this implementation with empirical results.

We begin by introducing concurrency in the context of CHR (Section 5.1), this is followed by a brief demonstration of parallel programming with CHR (Section 5.1.2). Next, we formally introduce our concurrent goal-based CHR semantics, $\| \mathcal{G}$. It forms the basis for an efficient parallel CHR implementation (Section 5.2). Section 5.3 presents a proof of its correspondence with the abstract CHR semantics. In [Lam, 2010] several subtle issues of the $\| \mathcal{G}$ semantics are highlighted.

Refining from [Sulzmann and Lam, 2008], we provide details of implementing a Parallel CHR system. Section 5.4 provides a quick review on existing CHR goal based implementations which describes sequential goal execution. We highlight our parallel CHR implementation in Haskell(GHC) (Section 5.5) and provide experiment results in Section 5.6.

Communication channel:

$$get \ @ \ Get(x), Put(y) \Longleftrightarrow x = y$$

$$\frac{\{Get(m), Put(1)\} \rightarrowtail_{get} \{m = 1\} \parallel \{Get(n), Put(8)\} \rightarrowtail_{get} \{n = 8\}}{\{Get(m), Put(1), Get(n), Put(8)\} \rightarrowtail^* \{m = 1, n = 8\}}$$

Greatest common divisor:

$$gcd1 \ @ \ Gcd(0) \Longleftrightarrow True$$
$$gcd2 \ @ \ Gcd(n) \backslash Gcd(m) \Longleftrightarrow m >= n \&\& n > 0 \mid Gcd(m - n)$$

$$\frac{\begin{array}{c}\{Gcd(3), Gcd(9)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(6)\} \\ \parallel \\ \{Gcd(3), Gcd(18)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(15)\}\end{array}}{\begin{array}{lll}\{Gcd(3), Gcd(9), Gcd(18)\} & \rightarrowtail_{gcd2,gcd2} & \{Gcd(3), Gcd(6), Gcd(15)\} \\ & \rightarrowtail^* & \{Gcd(3)\}\end{array}}$$
$$\overline{\{Gcd(3), Gcd(9), Gcd(18)\} \rightarrowtail^* \{Gcd(3)\}}$$

Figure 5.1: Communication channel and greatest common divisor

Due to the focus on Haskell, this chapter follows the Haskell rather than the Prolog syntax conventions. For example, variable names are lowercase characters.

5.1 CHR and Concurrency

The abstract CHR semantics is non-deterministic and highly concurrent. Rule instances can be applied concurrently as long as they do not interfere. By interfere, we mean that they simplify (delete) distinct constraints in a store. In other words, they do not content for the same resources by attempting to simplify the same constraints.

Figure 5.1 illustrates concurrency via two examples, communication buffer and greatest common divisor (Gcd). We indicate concurrent derivations via the symbol $\parallel$. Given derivation steps $\{Get(m), Put(1)\} \rightarrowtail_{get} \{m = 1\}$ and $\{Get(n), Put(8)\} \rightarrowtail_{get} \{n = 8\}$, we can straightforwardly combine both derivations, which leads to the final store $\{m = 1, n = 8\}$. Note that we have another possible final store $\{n = 1, m = 8\}$, that is deriveable from an initial store $\{Get(m), Put(1), Get(n), Put(8)\}$. The abstract CHR semantics is non-deterministic and can possibly yield more than one result for a particular program.

For Gcd, we show a more complex parallel composition: we combine the derivations $\{Gcd(3), Gcd(9)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(6)\}$ and $\{Gcd(3), Gcd(18)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(15)\}$ in a way that they share only propagated constraints (i.e. $Gcd(3)$). The resultant parallel derivation is consistent since the propagated components are not deleted.

5.1.1 Concurrency in the Abstract CHR Semantics

An important property in the CHR abstract semantics is *monotonicity*. Illustrated in Theorem 4, monotonicity of CHR execution guarantees that derivations of the CHR abstract semantics remain valid if we include a larger context (eg. $A \rightarrowtail^* B$ is valid under the additional context of constraints S, hence $A \uplus S \rightarrowtail^* B \uplus S$). This has been formally verified in [Abdennadher et al., 1999].

Theorem 4 (Monotonicity of CHR)**.** *For any sets of CHR constraints A,B and S, if* $A \rightarrowtail^* B$ *then* $A \uplus S \rightarrowtail^* B \uplus S$

An immediate consequence of monotonicity is that concurrent CHR executions as sketched above are sound in the sense that their effect can be reproduced using an appropriate sequential sequence of execution steps. Based on this observation, we can justify the concurrency rule (using an interleaving semantics approach),

$$(\textbf{Concurrency}) \quad \frac{S \uplus S_1 \rightarrowtail^* S \uplus S_2 \quad S \uplus S_3 \rightarrowtail^* S \uplus S_4}{S \uplus S_1 \uplus S_3 \rightarrowtail^* S \uplus S_2 \uplus S_4}$$

This rule essentially states that CHR derivations which affect different parts of the constraint store can be composable (i.e. joined as though that occur concurrently). In [Frühwirth, 2005], the above is referred to as "Strong Parallelism of CHR". However, we prefer to use the term "concurrency" instead of "parallelism". In our context, concurrency means to run a CHR program (i.e. a set of CHR rules) by using concurrent execution threads. A related, more detailed discussion on parallelism and concurrency for CHR can be found in [Frühwirth, 2009].

5.1.2 Parallel Programming in CHR

The example from [Frühwirth, 2009] in Figure 5.2 is a CHR encoding of the well-known merge sort algorithm. To sort a sequence of (distinct) elements $e_1, ..., e_m$ where m is a power of 2, we apply the rules to the initial constraint store $Merge(1, e_1), ..., Merge(1, e_m)$. Constraint $Merge(n, e)$ refers to a sorted sequence of numbers at level n whose smallest element is e. Constraint $Next(a, b)$ denotes that a is before b in the sorting order. Rule *merge2* initiates the merging of two sorted lists and creates a new sorted list at the next level. The actual merging is performed by rule *merge1*. Sorting of sublists belonging to different mergers can be performed simultaneously. See the example derivation in Figure 5.2 where we simultaneously sort the characters a, c, e, g and b, d, f, h.

For another example, consider the following CHR rules implementing a concurrent dictionary, whose concurrent *lookup* and *set* operations can occur in parallel as long as the operated keys are non-overlapping:

$$\begin{array}{ccl} lookup & @ & Entry(k1, v) \backslash Lookup(k2, x) \iff k1 == k2 \mid x = v \\ set & @ & Set(k1, v), Entry(k2, \_) \iff k1 == k2 \mid Entry(k2, v) \\ new & @ & NewEntry(k, v) \iff Entry(k, v) \end{array}$$

Constraint $Entry(k, v)$ represents a dictionary mapping of key k to value v. The CHR rule *lookup* models the action of looking up a key $k2$ in the dictionary, and assigning its value to v. Similarly, the CHR rule *set* represents the action of setting a new value v to the dictionary key k, while *new* creates new entries in the dictionary. Note that constraints $Lookup(k, x)$, $Set(k, v)$ and $newEntry(k, v)$ represent triggers to the respective actions. The following derivation illustrates non-overlapping dictionary operations:

$$\begin{array}{c} \{Lookup('a', x1), Entry('a', 1)\} \rightarrowtail \{x1 = 1, Entry('a', 1)\} \\ || \\ \{Lookup('b', x2), Entry('b', 2)\} \rightarrowtail \{x2 = 2, Entry('b', 2)\} \\ || \\ \{Set('c', 10), Entry('c', 3)\} \rightarrowtail \{Entry('c', 10)\} \\ \hline \begin{array}{cl} & \{Lookup('a', x1), Lookup('b', x2), Set('c', 10), Entry('a', 1), Entry('b', 2), Entry('c', 3)\} \\ \rightarrowtail^* & \{x1 = 1, x2 = 2, Entry('a', 1), Entry('b', 2), Entry('c', 10)\} \end{array} \end{array}$$

$$merge1 \ @ \ Next(x,a) \setminus Next(x,b) \iff a < b \mid Next(a,b)$$
$$merge2 \ @ \ Merge(n,a), Merge(n,b) \iff a < b \mid Next(a,b), Merge(n+1,a)$$

Shorthands: $N = Next$ and $M = Merge$

$$\begin{array}{rl}
 & M(1,a), M(1,c), M(1,e), M(1,g) \\
\rightarrowtail_{merge2} & M(2,a), M(1,c), M(1,e), N(a,g) \\
\rightarrowtail_{merge2} & M(2,a), M(2,c), N(a,g), N(c,e) \\
\rightarrowtail_{merge2} & M(3,a), N(a,g), N(c,e), N(a,c) \\
\rightarrowtail_{merge1} & M(3,a), N(a,c), N(c,g), N(c,e) \\
\rightarrowtail_{merge1} & M(3,a), N(a,c), N(c,e), N(e,g)
\end{array}$$

$$\Vert$$

$$\begin{array}{rl}
 & M(1,b), M(1,d), M(1,f), M(1,h) \\
\rightarrowtail^* & M(3,b), N(b,d), N(d,f), N(f,h)
\end{array}$$

$$\begin{array}{rl}
 & M(3,a), N(a,c), N(c,e), N(e,g), M(3,b), N(b,d), N(d,f), N(f,h) \\
\rightarrowtail_{merge2} & M(4,a), N(a,c), N(a,b), N(c,e), N(e,g), N(b,d), N(d,f), N(f,h) \\
\rightarrowtail_{merge1} & M(4,a), N(a,b), N(b,c), N(c,e), N(e,g), N(b,d), N(d,f), N(f,h) \\
\rightarrowtail_{merge1} & M(4,a), N(a,b), N(b,c), N(c,d), N(c,e), N(e,g), N(d,f), N(f,h) \\
\rightarrowtail_{merge1} & M(4,a), N(a,b), N(b,c), N(c,d), N(d,e), N(e,g), N(d,f), N(f,h) \\
\rightarrowtail_{merge1} & M(4,a), N(a,b), N(b,c), N(c,d), N(d,e), N(e,f), N(e,g), N(f,h) \\
\rightarrowtail_{merge1} & M(4,a), N(a,b), N(b,c), N(c,d), N(d,e), N(e,f), N(f,g), N(f,h) \\
\rightarrowtail_{merge1} & M(4,a), N(a,b), N(b,c), N(c,d), N(d,e), N(e,f), N(f,g), N(g,h)
\end{array}$$

$$\begin{array}{rl}
 & M(1,a), M(1,c), M(1,e), M(1,g), M(1,b), M(1,d), M(1,f), M(1,h) \\
\rightarrowtail^* & M(4,a), N(a,b), N(b,c), N(c,d), N(d,e), N(e,f), N(f,g), N(g,h)
\end{array}$$

Figure 5.2: Merge sort

In the last example, we implement the parallel programming framework *map-reduce* in CHR:

$$\begin{array}{rcl}
map1 & @ & Map((x:xs),m,r) \iff Work(x,m,r), Map(xs,m,r) \\
map2 & @ & Map([\,],_,_) \iff True \\
work & @ & Work(x,m,r) \iff Reduce([m(x)],r) \\
reduce & @ & Reduce(xs1,r), Reduce(xs2,_) \iff Reduce(r(xs1,xs2),r)
\end{array}$$

We assume that m and r are higher-order functions representing the abstract *map* and *reduce* functions. The constraint $Map(xs,m,r)$ initiates the *map1* rule which maps the function m onto each element in xs. Each application of m is represented by $Work(x,m,r)$ and the actual application $m(x)$ is implemented by the rule *work*, producing the results $Reduce(xs,r)$. The rule *reduce* models the reduce step, combining the results in the manner specified by reduce function r[1] When CHR rewritings are exhaustively applied, the store will have a single $Reduce(xs,r)$ constraint where xs is the final result. Note that the

[1]For simplicity, we assume a simple setting, where the ordering of elements need not be preserved.

Notations:

$\uplus$	Multi-set union
$\cup$	Set union
$\models$	Theoretic entailment
ϕ	Substitution
$\overline{a}$	Set/List of a's

CHR Syntax:

Functions	$f ::= + \mid >\mid \&\& \mid ...$
Constants	$v ::= 1 \mid true \mid ...$
Terms	$t ::= x \mid f\ \overline{t}$
Predicates	$p ::= Get \mid Put \mid ...$
Equations	$e ::= t = t$
CHR Constraints	$c ::= p(\overline{t})$
Constraints	$b ::= e \mid c$
CHR Guards	$t_g ::= t$
CHR Heads	$H ::= \overline{c}$
CHR Body	$B ::= \overline{b}$
CHR Rule	$R ::= r\ @\ H \setminus H \Longleftrightarrow t_g \mid B$
CHR Program	$\mathcal{P} ::= \overline{R}$
Num Constraint	$nc ::= c\#i$
Goal Constraint	$g ::= c \mid e \mid nc$
Stored Constraint	$sc ::= nc \mid e$
CHR Num Store	$Sn ::= \overline{sc}$
CHR Goals	$G ::= \overline{g}$
CHR State	$\sigma ::= \langle G, Sn \rangle$
Side Effects	$\delta ::= Sn \setminus Sn$

Figure 5.3: CHR Goal-based Syntax

concurrent CHR semantics models the parallelism of the map reduce framework: multiple $Work(x, m, r)$ constraints are free to be applied to the *work* rule concurrently, while non-overlapping pairs of $Reduce(xs, r)$ can be combined by the *reduce* rule concurrently.

Note that in the examples above, the CHR rules here declaratively defines the synchronization patterns of the constraints representing concurrent processes, while the concurrent CHR semantics abstracts away the actual details of the synchronization. To execute such programs to scale with multi-core systems, we will require an implementation of the CHR concurrent semantics that actually executes multiple CHR rewritings in parallel. We will provide details of such an implementation in Chapter 5.4.

5.2 Concurrent Goal-Based Refined CHR Semantics

We present the formal details of the concurrent goal-based CHR semantics. Figure 5.3 describes the necessary syntactic extensions. Because constraints in the store now have unique identifiers, we treat the store as a set (as opposed to a multiset) and use set union $\cup$. Goals are still treated as multi-sets because they can contain multiple copies of (un-numbered) CHR constraints. Please note that we will use lower-case identifiers for variables, upper-case identifiers for constants[2] Note that we will only consider CHR

[2]Advocates of logic constraint programming should have noticed this "abnormally". We sincerely apologize, but insist that this is for consistency with our Haskell formulation of CHR in Chapter 5.4

$$\textbf{(Solve)} \quad \frac{W = WakeUp(e, Sn)}{\langle \{e\} \uplus G \mid Sn\rangle \overset{W\backslash\{\}}{\rightarrowtail_{\mathcal{G}}} \langle W \uplus G \mid \{e\} \cup Sn\rangle}$$

$$\textbf{(Activate)} \quad \frac{i \text{ is a fresh identifier}}{\langle \{c\} \uplus G \mid Sn\rangle \overset{\{\}\backslash\{\}}{\rightarrowtail_{\mathcal{G}}} \langle \{c\#i\} \uplus G \mid \{c\#i\} \cup Sn\rangle}$$

$$\textbf{(Simplify)} \quad \frac{\begin{array}{c}(r \ @\ H'_P \backslash H'_S \iff t_g \mid B') \in \mathcal{P} \text{ such that} \\ \exists \phi \quad Eqs(Sn) \models \phi \wedge t_g \quad \phi(H'_P) = DropIds(H_P) \\ \phi(H'_S) = \phi(\{c\} \uplus DropIds(H_S)) \quad \delta = H_P \backslash \{c\#j\} \cup H_S\end{array}}{\begin{array}{c}\langle \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup H_S \cup Sn\rangle \\ \overset{\delta}{\rightarrowtail_{\mathcal{G}}} \quad \langle \phi(B') \uplus G \mid H_S \cup Sn\rangle\end{array}}$$

$$\textbf{(Propagate)} \quad \frac{\begin{array}{c}(r \ @\ H'_P \backslash H'_S \iff t_g \mid B') \in \mathcal{P} \text{ such that} \\ \exists \phi \quad Eqs(Sn) \models \phi \wedge t_g \quad \phi(H'_S) = DropIds(H_S) \\ \phi(H'_P) = \phi(\{c\} \uplus DropIds(H_P)) \quad \delta = \{c\#j\} \cup H_P \backslash H_S\end{array}}{\begin{array}{c}\langle \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup H_S \cup Sn\rangle \\ \overset{\delta}{\rightarrowtail_{\mathcal{G}}} \quad \langle \phi(B') \uplus \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup Sn\rangle\end{array}}$$

$$\textbf{(Drop)} \quad \frac{\text{(Simplify) and (Propagate) does not apply on } c\#j \text{ in } Sn}{\langle \{c\#j\} \uplus G \mid Sn\rangle \overset{\{\}\backslash\{\}}{\rightarrowtail_{\mathcal{G}}} \langle G \mid Sn\rangle}$$

where
$$\begin{array}{lcl} Eqs(S) & = & \{e \mid e \in S, e \text{ is an equation}\} \\ DropIds(Sn) & = & \{c \mid c\#i \in Sn\} \uplus \{e \mid e \in Sn, e \text{ is an equation}\} \\ WakeUp(e, Sn) & = & \{c\#i \mid c\#i \in Sn \wedge \phi \text{ m.g.u. of } Eqs(Sn) \wedge \\ & & \theta \text{ m.g.u. of } Eqs(Sn \cup \{e\}) \wedge \phi(c) \neq \theta(c)\} \end{array}$$

Figure 5.4: Goal-Based CHR Semantics (Single-Step Execution $\overset{\delta}{\rightarrowtail_{\mathcal{G}}}$)

rules with non-empty simplification heads (i.e. no pure propagation rules). The actual semantics is given in two parts. Figure 5.4 describes the single-step execution part whereas Figure 5.5 introduces the concurrent execution part. The first part is a generalization of the refined CHR semantics as given in [Duck, 2005] whereas the second (concurrent) part is novel.

We first discuss the single-step derivation steps in Figure 5.4. A derivation step $\sigma \overset{\delta}{\rightarrowtail_{\mathcal{G}}} \sigma'$ maps the CHR state σ to σ' with some side-effect δ. δ represents the constraints that where propagated or simplified during rule application. Hence derivation steps that do not involve rule application ((Activate) and (Drop)) contains no side-effects (i.e. $\{\}\backslash\{\}$). We will omit side-effects δ as and when it is not relevant to our discussions. We ignore the (Solve) step for the moment. In (Activate), we activate a goal CHR constraint by assigning it a fresh unique identifier and adding it to the store. Rewrite rules are executed in steps (Simplify) and (Propagate). We distinguish if the rewrite rule is executed on a simplified or propagated active (goal) constraint $c\#i$. For both cases, we seek for the missing partner constraints in the store for some matching substitution ϕ. The auxiliary function $DropIds$ ignores the unique identifiers of numbered constraints. They don't matter when finding a rule head match. The guard t_g must be entailed by the primitive (here equations) store constraints under the substitution ϕ.

In case of a simplified goal, step (Simplify), we apply the rule instance of r by deleting

$$\textbf{(Lift)} \qquad \frac{\langle G \mid Sn\rangle \overset{\delta}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid Sn'\rangle}{\langle G \mid Sn\rangle \overset{\delta}{\rightarrowtail_{||\mathcal{G}}} \langle G' \mid Sn'\rangle}$$

$$\textbf{(Goal Concurrency)} \qquad \frac{\begin{array}{c}\langle G_1 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_1}{\rightarrowtail_{||\mathcal{G}}} \langle G_1' \mid H_{S2} \cup S\rangle \\ \langle G_2 \mid H_{S1} \cup H_{S2} \cup S\rangle \overset{\delta_2}{\rightarrowtail_{||\mathcal{G}}} \langle G_2' \mid H_{S1} \cup S\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq S \quad H_{P2} \subseteq S \quad \delta = H_{P1} \cup H_{P2} \backslash H_{S1} \cup H_{S2}\end{array}}{\begin{array}{c}\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \cup H_{S2} \cup S\rangle \\ \overset{\delta}{\rightarrowtail_{||\mathcal{G}}} \quad \langle G_1' \uplus G_2' \uplus G \mid S\rangle\end{array}}$$

$$\textbf{(Closure)} \qquad \frac{\sigma \overset{\delta}{\rightarrowtail_{||\mathcal{G}}} \sigma'}{\sigma \overset{*}{\rightarrowtail_{||\mathcal{G}}} \sigma'} \qquad \frac{\sigma \overset{\delta}{\rightarrowtail_{||\mathcal{G}}} \sigma' \quad \sigma' \overset{*}{\rightarrowtail_{||\mathcal{G}}} \sigma''}{\sigma \overset{*}{\rightarrowtail_{||\mathcal{G}}} \sigma''}$$

Figure 5.5: Goal-Based CHR Semantics (Concurrent Part $\overset{\delta}{\rightarrowtail_{||\mathcal{G}}}$)

all simplified matching constraints H_S and adding the rule body instance $\phi(B)$ into the goals. Since $c\#i$ is simplified, we drop $c\#i$ from the goals as it does not exist in the store any more. In case of a propagated goal, step (Propagate), $c\#i$ remains in the goal set as well as in the store and thus can possibly fire further rules instances. For both (Simplify) and (Propagate) derivation step, say $\sigma \overset{H_P \backslash H_S}{\rightarrowtail_{\mathcal{G}}} \sigma'$, we record as side-effect the numbered constraints in the store that were propagated (H_P) or simplified (H_S) during the derivation step. We will elaborate on the purpose of side-effects when we introduce the concurrent part of the semantics.

In step (Drop), we remove an active constraint from the set of goals, if the constraint failed to trigger any CHR rule.

Rule (Solve) moves an equation goal e into the store and *wakes up* (reactivates) any numbered constraint in the store which can possibly trigger further CHR rules due to the presence of e. Here is a simple example to show why reactivation is necessary.

$r1 \ @ \ A(x), B(x) \Longleftrightarrow C(x)$

$$\begin{array}{lll} & & \langle \{a = 2\} \mid \{A(a)\#1, B(2)\#2\}\rangle \\ \text{(Solve)} & \overset{\{A(2)\#1\}\backslash\{\}}{\rightarrowtail_{\mathcal{G}}} & \langle \{A(2)\#1\} \mid \{A(2)\#1, B(2)\#2, a = 2\}\rangle \\ \text{(Simp } r1\text{)} & \overset{\{\}\backslash\{A(2)\#1, B(2)\#2\}}{\rightarrowtail_{\mathcal{G}}} & \langle \{C(2)\} \mid \{a = 2\}\rangle \\ \ldots & & \end{array}$$

For clarity, we normalize all constraints in the store once an equation is added. Prior to addition of $a = 2$, $A(a)\#1, B(2)\#2$ cannot fire rule $r1$. After adding $a = 2$ however, we can normalize $A(a)\#1$ to $A(2)\#2$, which can now fire $r1$ with $B(2)\#2$. To guarantee exhaustive rule firings, we reactivate $A(2)\#2$ by adding it back to the set of goals. $WakeUp(e, Sn)$ represents a conservative approximation of the to be reactivated constraints [Duck, 2005]. Note we treat reactivated constraints as propagated constraints in the side-effects.

Figure 5.5 presents the concurrent part of the goal-based operational semantics. In the (Lift) step, we turn a sequential goal-based derivation into a concurrent derivation.

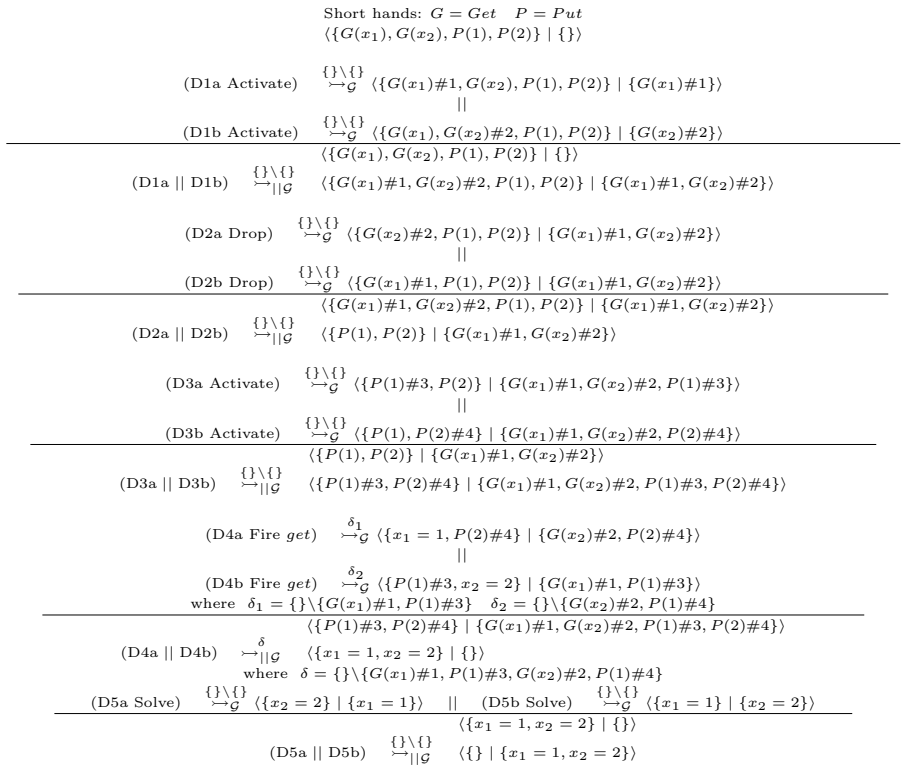


Figure 5.6: Example of concurrent goal-based CHR derivation

Note that side-effects are retained. Step (Goal Concurrency) joins together two concurrent derivations operating on a shared store, if their rewriting side-effects δ_1 and δ_2 are non-overlapping as defined below.

Definition 5.2.1 (Non-overlapping Rewriting Side-Effects)**.** *Two rewriting side-effects $\delta_1 = H_{P1} \backslash H_{S1}$ and $\delta_2 = H_{P2} \backslash H_{S2}$ are said to be non-overlapping, if and only if $H_{S1} \cap (H_{P2} \cup H_{S2}) = \{\}$ and $H_{S2} \cap (H_{P1} \cup H_{S1}) = \{\}$*

Concurrent derivations with non-overlapping side-effects essentially simplify distinct constraints in the store, as well as propagate constraints which are not simplified by one another. The (Goal Concurrency) step expresses non-overlapping side-effects by structurally enforcing that simplified constraints H_{S1} and H_{S2} match distinct parts of the store, while propagated constraints H_{P1} and H_{P2} are found in the shared part of the store S not modified by both concurrent derivations. In the resulting concurrent derivation, the side-effects δ_1 and δ_2 are composed by the union of the propagate and simplify components respectively, forming δ.

The (Closure) step defines transitive application of the concurrent goal-based derivation. Because side-effect labels are only necessary for the (Goal Concurrency) step, we drop the side-effects in transitive derivations.

Figure 5.6 shows a sample concurrent goal-based CHR derivation. We assume two concurrent threads, referred to as a and b, each thread executes the standard goal-based derivation steps. The novelty is that each goal-based derivation step $\overset{\delta}{\rightarrowtail_{\mathcal{G}}}$ now records its

effect on the store. The effect δ represents the sets of constraints in the store which were propagated or simplified. Goal-based derivation steps can be executed concurrently if their effects are not in conflict.

Each thread activates one of the two *Get* goals (Steps $D1a$ and $D1b$). Since both steps involve no rule application, side-effects are empty ($\{\}\backslash\{\}$). Both steps are executed concurrently denoted by the concurrent derivation step $(D1a \mid\mid D2a) \overset{\{\}\backslash\{\}}{\rightarrowtail}_{||\mathcal{G}}$. Concurrent goal-based execution threads operate on a shared store and their effects will be immediately made visible to other threads. This is important to guarantee exhaustive rule firings.

In the second step ($D2a \mid\mid D2b$), both active goals are dropped because there is no complete match for any rule head yet. Next, steps $D3a$ and $D3b$ activate the last two goal constraints, $Put(1)$ and $Put(2)$. Each active constraint can match with either of the two *Get* constraints in the store. We assume that active constraint $Put(1)\#3$ in step $D4a$ matches with $Get(x_1)\#1$, while $Put(2)\#4$ in step $D4b$ matches with $Get(x_2)\#2$, corresponding to the side-effects δ_1 and δ_2. This guarantees that steps $D4a$ and $D4b$ operates on different (non-conflicting) parts of the store. Thus, we can execute them concurrently which yields step ($D4a \mid\mid D4b$). Their side-effects are combined as δ. Finally, in step ($D5a \mid\mid D5b$) we concurrently solve the two remaining equations by adding them into the store and we are done.

5.3 Correspondence Results

The correctness of our concurrent goal-based semantics is established by showing that all concurrent derivations can be replicated by sequential goal-based executions. We also prove that there is a correspondence between our goal-based CHR semantics with the abstract CHR semantics. This proof generalizes from [Duck, 2005] which shows a correspondence between the refined CHR operational semantics and abstract semantics.

We formally verify that the concurrent goal-based semantics is in exact correspondence to the abstract CHR semantics when it comes to termination and exhaustive rule firings. Detailed proofs are given in [Lam, 2010]. In the following sections, we provide key lemmas and proof sketches.

5.3.1 Formal Definitions

We first introduce some elementary definitions before stating the formal results.

The first two definitions concern the abstract CHR semantics. A store is final if no further rules are applicable.

Definition 5.3.1 (Final Store)**.** *A store S is known as a final store, denoted $Final_{\mathcal{A}}(S)$ if and only if no more CHR rules applies on it (i.e. $\neg\exists S'$ such that $S \rightarrowtail_{\mathcal{A}} S'$).*

A CHR program terminates if all derivations lead to a final store in a finite number of states.

Definition 5.3.2 (Terminating CHR Programs)**.** *A CHR program $\mathcal{P}$ is said to be terminating, if and only if for any CHR store S, there exists no infinite derivation paths from S, via the program $\mathcal{P}$.*

Next, we introduce some definitions in terms of the goal-based semantics. In an initial state, all constraints are goals and the store is empty. Final states are states which no longer have any goals. We will prove the exhaustiveness of the goal-based semantics by

proving a correspondence between final stores in the abstract semantics and final states of the goal-based semantics

Definition 5.3.3 (Initial and Final CHR States)**.** *An initial CHR state is a CHR state of the form* $\langle G \mid \{\}\rangle$ *where* G *contains no numbered constraints (c#n), while a final CHR state is of the form* $\langle \{\} \mid Sn\rangle$

A state is reachable if there exists a (sequential) goal-based sequence of derivations to this state. We write $\rightarrowtail_{\mathcal{G}}^{*}$ to denote the transitive closure of $\rightarrowtail_{\mathcal{G}}$.

Definition 5.3.4 (Sequentially Reachable CHR states)**.** *For any CHR program* $\mathcal{P}$*, a CHR state* $\langle G' \mid Sn'\rangle$ *is said to be sequentially reachable by* $\mathcal{P}$ *if and only if there exists some initial CHR state* $\langle G \mid \{\}\rangle$ *such that* $\langle G \mid \{\}\rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G' \mid Sn'\rangle$.

5.3.2 Correspondence of Derivations

We build a correspondence between the abstract semantics and the concurrent goal-based semantics. We begin with Theorem 5, which states the correspondence of the (sequential) goal-based semantics.

Theorem 5 (Correspondence of Sequential Derivations)**.** *For any reachable CHR state* $\langle G \mid Sn\rangle$*, CHR state* $\langle G' \mid Sn'\rangle$ *and CHR program* $\mathcal{P}$*,*

$$\begin{array}{ll} \textit{if} & \langle G \mid Sn\rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G' \mid Sn'\rangle \\ \textit{then} & (NoIds(G) \uplus DropIds(Sn)) = (NoIds(G') \uplus DropIds(Sn')) \quad \vee \\ & (NoIds(G) \uplus DropIds(Sn)) \rightarrowtail_{\mathcal{A}}^{*} (NoIds(G') \uplus DropIds(Sn')) \end{array}$$

where $NoIds = \{c \mid c \in G, c \textit{ is a CHR constraint}\} \uplus \{e \mid e \in G, e \textit{ is an equation}\}$

The above result guarantees that any sequence of sequential goal-based derivations starting from a reachable CHR state either yields equivalent CHR abstract stores (due to goal-based behaviour not captured by the abstract semantics, namely (Solve) (Activate), (Drop)) or corresponds to a derivation in the abstract semantics (due to rule application). A goal-based semantics state $\langle G \mid Sn\rangle$ is related to an abstract semantics store by removing all numbered constraints in G and union it with constraints in Sn without their identifiers. The theorem and its proof is a generalization of an earlier result given in [Duck, 2005].

We formalize the observation that the goal context can be extended without interfering with previous goal executions.

Lemma 5.3.5 (Monotonicity of Goals in Goal-based Semantics)**.** *For any goals G,G′ and G″ and CHR store Sn and Sn′, If* $\langle G \mid Sn\rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G' \mid Sn'\rangle$ *then* $\langle G \uplus G'' \mid Sn\rangle \rightarrowtail_{\mathcal{G}}^{*} \langle G' \uplus G'' \mid Sn'\rangle$.

Next, we state that given any goal-based derivation with side-effects δ, we can safely ignore any constraints (represented by S_2) in the store which is not part of δ.

Lemma 5.3.6 (Isolation of Goal-based Derivations)**.**

$$\begin{array}{ll} If & \langle G \mid H_P \cup H_S \cup S_1 \cup S_2\rangle \overset{H_P \setminus H_S}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid H_P \cup S_1' \cup S_2\rangle \\ then & \langle G \mid H_P \cup H_S \cup S_1\rangle \overset{H_P \setminus H_S}{\rightarrowtail_{\mathcal{G}}} \langle G' \mid H_P \cup S_1'\rangle \end{array}$$

Lemma 5.3.6 can be straight-forwardly extended to multiple derivation steps. This is stated in Lemma 5.3.7.

Lemma 5.3.7 (Isolation of Transitive Goal-based Derivations)**.**

$$\begin{array}{ll} If & \langle G \mid H_P \cup H_S \cup S_1 \cup S_2 \rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid H_P \cup S'_1 \cup S_2 \rangle \\ & \textit{with side-effects } \delta = H_P \backslash H_S \\ then & \langle G \mid H_P \cup H_S \cup S_1 \rangle \rightarrowtail^*_{\mathcal{G}} \langle G' \mid H_P \cup S'_1 \rangle \end{array}$$

The next lemma states that any concurrent derivation starting from a reachable CHR state can be replicated by a sequence of sequential goal-based derivations. Lemma 5.3.8 is the first step to prove the correspondence of concurrent goal-based derivations.

Lemma 5.3.8 (Sequential Reachability of Concurrent Derivation Steps)**.** *For any sequentially reachable CHR state σ, CHR state σ' and rewriting side-effects δ if $\sigma \overset{\delta}{\rightarrowtail}_{||\mathcal{G}} \sigma'$ then σ' is sequentially reachable, $\sigma \rightarrowtail^*_{\mathcal{G}} \sigma'$ with side-effects δ.*

Proof. (Sketch) We can always reduce k mutually non-overlapping concurrent derivations into several applications of the (Goal Concurrency) step. Hence we can prove Lemma 5.3.8 by structural induction over the concurrent goal-based derivation steps (Lift) and (Goal Concurrency) where we use Lemmas 5.3.5 and 5.3.7 to show that concurrent derivations can always be replicated by a sequence of sequential goal-based derivations. □

Theorem 6 (Sequential Reachability of Concurrent Derivations)**.** *For any initial CHR state σ, CHR state σ' and CHR Program $\mathcal{P}$, if $\sigma \rightarrowtail^*_{||\mathcal{G}} \sigma'$ then $\sigma \rightarrowtail^*_{\mathcal{G}} \sigma'$.*

The above can be directly proven from Lemma 5.3.8 by converting each single step concurrent derivation into a sequence of sequential derivations, and showing their composibility.

From Theorem 5 and 6, we have the following corollary, which states the correspondence between concurrent goal-based CHR derivations and abstract CHR derivations.

Corollary 5.3.9 (Correspondence of Concurrent Derivations)**.** *For any reachable CHR state $\langle G \mid Sn \rangle$, CHR state $\langle G' \mid Sn' \rangle$ and CHR program $\mathcal{P}$,*

$$\begin{array}{ll} if & \langle G \mid Sn \rangle \rightarrowtail^*_{||\mathcal{G}} \langle G' \mid Sn' \rangle \\ then & (NoIds(G) \uplus DropIds(Sn)) = (NoIds(G') \uplus DropIds(Sn')) \quad \vee \\ & (NoIds(G) \uplus DropIds(Sn)) \rightarrowtail^*_{\mathcal{A}} (NoIds(G') \uplus DropIds(Sn')) \end{array}$$

where $NoIds = \{c \mid c \in G, c \text{ is a CHR constraint}\} \uplus \{e \mid e \in G, e \text{ is an equation}\}$

5.3.3 Correspondence of Termination

We show that all derivations from a initial state to final states in the concurrent goal-based semantics corresponds to some derivation from a store to a final store in the abstract semantics. We first define rule head instances:

Definition 5.3.10 (Rule head instances)**.** *For any CHR state $\sigma = \langle G, Sn \rangle$ and CHR program $\mathcal{P}$, any $(H_P \cup H_S) \subseteq Sn$ is known as a rule head instance of σ, if and only if $\exists (r \ @ \ H'_P \backslash H'_P \iff t_g \mid B) \in \mathcal{P}$,$\exists \phi \ Eqs(Sn) \models \phi \wedge t_g$ and $\phi(H'_P \uplus H'_S) = DropIds(H_P \cup H_S)$.*

Definition 5.3.11 (Active rule head instances)**.** *For any CHR state $\sigma = \langle G, Sn \rangle$ and CHR program $\mathcal{P}$, a rule head instance H of σ is said to be* active *if and only if there exists at least one $c\#i \in G$ such that $c\#i \in H$.*

Rule head instances (Definition 5.3.10) are basically minimal subsets of the store which matches a rule head. Active rule head instance (Definition 5.3.11) additional have at least one of its numbered constraint $c\#i$ in the goals as well. Therefore, by the definition of the goal-based semantics, active rule head instances will eventually be triggered by either the (Simplify) or (Propagate) derivation steps.

Lemma 5.3.12 (Rule instances in reachable states are always active)**.** *For any reachable CHR state $\langle G \mid Sn\rangle$, any rule head instance $H \subseteq Sn$ must be active. i.e. $\exists c\#i \in H$ such that $c\#i \in G$.*

Lemma 5.3.12 shows that all rule head instances in reachable states are always active. This means that by applying the semantics steps in any way, we must eventually apply the rule head instances as long as all its constraints remain in the store.

Theorem 7 states that termination of a concurrent goal-based derivation corresponds to termination in the abstract semantics. This is of course, provided that the CHR program is terminating.

Theorem 7 (Correspondence of Termination)**.** *For any initial CHR state $\langle G, \{\}\rangle$, final CHR state $\langle \{\}, Sn\rangle$ and terminating CHR program $\mathcal{P}$,*

$$\begin{array}{l} \textit{if } \langle G \mid \{\}\rangle \rightarrowtail^*_{\parallel\mathcal{G}} \langle \{\} \mid Sn\rangle \\ \textit{then } G \rightarrowtail^*_{\mathcal{A}} DropIds(Sn) \textit{ and } Final_{\mathcal{A}}(DropIds(Sn)) \end{array}$$

We prove this theorem by first using Theorem 6 which guarantees that a concurrent goal-based derivation from an initial state to a final state corresponds to some abstract semantics derivation. We next show that final states corresponds to final stores in the abstract semantics. This is done by contradiction, showing that assuming otherwise contradicts with Lemma 5.3.12.

5.4 Implementation of CHR, a Quick Review

In the execution of CHR goals, rule head matching is essentially the most technically complex and computationally intensive procedure that is involved. As such, any practical implementation of goal-based CHR execution must include a highly efficient rule-head matching routine. Recall the (simplify) derivation step of the concurrent goal-based semantics $\parallel \mathcal{G}$:

$$\textbf{(Simplify)} \quad \frac{\begin{array}{c}(r \ @\ H'_P \backslash H'_S \iff t_g \mid B') \in \mathcal{P} \text{ such that} \\ \exists\phi \quad Eqs(Sn) \models \phi \wedge t_g \quad \phi(H'_P) = DropIds(H_P) \\ \phi(H'_S) = \phi(\{c\} \uplus DropIds(H_S)) \quad \delta = H_P \backslash \{c\#j\} \cup H_S\end{array}}{\begin{array}{rl} & \langle \{c\#j\} \uplus G \mid \{c\#j\} \cup H_P \cup H_S \cup Sn\rangle \\ \overset{\delta}{\rightarrowtail}_{\mathcal{G}} & \langle \phi(B') \uplus G \mid H_S \cup Sn\rangle \end{array}}$$

This, as well as the (propagate) derivation step, models CHR rewritings in a declarative manner. But operationally, it specifies little about how the actual matching as well as searching for constraints is done. For instance, the premise of the derivation step simply states that given the goal $c\#j$, there must exist some constraints H_S and H_P in the constraint store that matches with the rule heads for this derivation step to be possible, but not exactly how such constraints in the store are located or how they are selected. In this section, we will provide more details on this problem which we will refer to as the *CHR goal-based matching problem*.

Non-linearized CHR Rule:

$$r1 \ @ \ A(1, x) \setminus B(x, y), C(z) \Longleftrightarrow y > z \mid D(x, y, z)$$

Linearized CHR Rule:

$$r1 \ @ \ A(1, x1) \setminus B(x2, y), C(z) \Longleftrightarrow y > z \wedge x1 = x2 \mid D(x1, y, z)$$

Figure 5.7: Linearizing CHR Rules

CHR Syntax:

Constants	v	::=	$1 \mid true \mid ...$
Terms	t	::=	$x \mid f\ \overline{t}$
Predicates	p	::=	$Get \mid Put \mid ...$
Equations	e	::=	$t = t$
CHR Constraints	c	::=	$p(\overline{t})$
Constraints	b	::=	$e \mid c$
CHR Guards	t_g	::=	t
CHR Heads	H	::=	$\overline{c}$
CHR Body	B	::=	$\overline{b}$
CHR Rule	R	::=	$r \ @ \ H \setminus H \Longleftrightarrow t_g \mid B$

CHR Goal-Based Rule Compilation:

Rewrite Type	rw	::=	$S \mid P$
Match Task	mt	::=	$Goal\ rw\ c \mid Lookup\ rw\ c \mid Guard\ t_g$
Match Task Sequence	mts	::=	$\overline{mt}$
Rule Compilation	occ	::=	(mts, B)

Figure 5.8: CHR Goal-Based Rule Compilation

5.4.1 CHR Goal-Based Rule Compilation

We highlight a compilation scheme for CHR rules which encodes CHR rules as a list of search tasks that locates a complete rule-head match, and a set of body constraints. This CHR compilation scheme, which we shall refer to as the *CHR Goal-based Rule Compilation*, is comparable with those used in existing CHR systems [Holzbaur et al., 2005].

For convenience, we assume that rule heads are linear. That is, each variable occurs at most once in a constraint in the rule head. It is straightforward to linearize CHR rules. For instance, Figure 5.7 shows the CHR rule $r1$ in its non-linearized and linearized form respectively.

Figure 5.8 shows the formal description of CHR goal-based rule compilations. For convenience, we also include the relevant fragment of the CHR syntax, shown earlier in Figure 5.3. The idea is to compiled a CHR rule, into a set of CHR goal-based rule compilations, where each uniquely corresponds to a rule head of the CHR rule. Each rule compilation is essentially a tuple that represents the sequence of *match tasks* to be executed when a goal is matched to its associated rule head, and a set of constraints

which represents the rule body. A match task specifies one of the three type of nodes, matching a goal (*Goal*), looking for a specific partner constraint (*Lookup*) or checking a guard condition (*Guard*). Each *Goal* or *Lookup* task is annotated by a *rewrite type* which distinguishes whether its goal/partner constraint is to be simplified (S) or propagated (P).

We illustrate this compilation scheme by example (A formal treatment to the compilation scheme is detail elsewhere [Duck, 2005, Schrijvers, 2005]). Let's consider our running example, rule $r1$ and its corresponding CHR goal-based rule compilations:

$$r1 \ @ \ A(1,x1) \setminus B(x2,y), C(z) \Longleftrightarrow y > z \wedge x1 = x2 \mid D(x1,y,z)$$

$$\begin{aligned} mts1 &= [Goal\ P\ A(1,x1), Lookup\ S\ B(x2,y), Lookup\ S\ C(z), Guard\ (y > z \wedge x1 = x2)] \\ mts2 &= [Goal\ S\ B(x2,y), Lookup\ P\ A(1,x1), Lookup\ S\ C(z), Guard\ (y > z \wedge x1 = x2)] \\ mts3 &= [Goal\ S\ C(z), Lookup\ P\ A(1,x1), Lookup\ S\ B(x2,y), Guard\ (y > z \wedge x1 = x2)] \end{aligned}$$

$$comp = \{(mts1, \{D(x1,y,z)\}), (mts2, \{D(x1,y,z)\}), (mts3, \{D(x1,y,z)\})\}$$

Rule $r1$ is compiled into three match tasks, namely $mts1$, $mts2$ and $mts3$, which corresponds to rule heads $A(1,x1)$, $B(x2,y)$ and $C(z)$. For instance, $mts1$ represents the match tasks for executing goals that matches with the rule head $A(1,x1)$, which involves looking for a partner constraint $B(x1,y)$ and then $C(z)$ and finally checking the guard constraints. This match task generates match trees like the one seen in Figure 5.9. Note that all well-formed match tasks have a leading *Goal* task.

5.4.2 CHR Goal-Based Lazy Matching

CHR goal-based matching is essentially a *lazy* matching problem. As opposed to *eagerly* matching all rule head instances in a given constraint store, for each CHR goal, we wish only to locate and execute rule head instances on demand. In essence, this matching problem involves some form of search routine which starts from a CHR goal and searches for matching constraints in the CHR store. This goal-based lazy matching routine is essentially encoded by the goal-based rule compilations discussed in the previous section.

Let's consider the CHR rule r from Figure 5.7. We model the search space of such matching problems via *match trees*. The particular match tree shown in Figure 5.9 represents the search space of the constraint matching problem for rule $r1$, triggered by the execution of the goal $A(1,x)$ (Match tasks $mts1$ from Section 5.4.1). Given the *Goal* node $A(1,x)$, we seek for constraints in the store matching rule heads $B(x,y)$ and $C(z)$, in this particular order[3]. For instance, the root (*Goal*) node $A(1,2)\#1$ has two child nodes, namely *Lookup* $B(2,10)\#2$ and $B(2,8)\#3$, each representing possible matches of $B(x,y)$ under the substitution $\{2/x, 10/y\}$ and $\{2/x, 8/y\}$ respectively. *Simp* and *Prop* tokens simply indicates if the constraint is to be simplified or propagated. *Guard* nodes represents the checking of CHR rule guards. Successful leaf nodes contain the complete rule head match which corresponds to all rule heads along the path from the root to the leaf node. By successful, we mean that the guard constraint is satisfied. Note that a complete specification of the matching problem for CHR rule r would include two other match trees, each of which specifies the matching problem starting from the each of the other two rule heads ($B(x,y)$ and $C(z)$).

The match tree in Figure 5.9 specifies four possible rule head instances (also referred to as *successful matches*). However, it is not possible to fire all of them together. This is because some of the matches are likely to contain overlapping rule heads. Note that for

[3]Note we can similarly have it in the order $C(z)$ then $B(x,y)$, but the abstract CHR semantics leaves this choice open. This flexibility allows us to use known CHR optimizations like optimal join-ordering

A CHR simpagation rule and Constraint Store:

$$r \ @ \ A(1,x) \setminus B(x,y), C(z) \Leftrightarrow y > z \mid D(x,y,z)$$
$$\{A(1,2)\#1, B(2,10)\#2, B(2,8)\#3, C(5)\#4, C(6)\#5, C(12)\#6\}$$

Match tree:

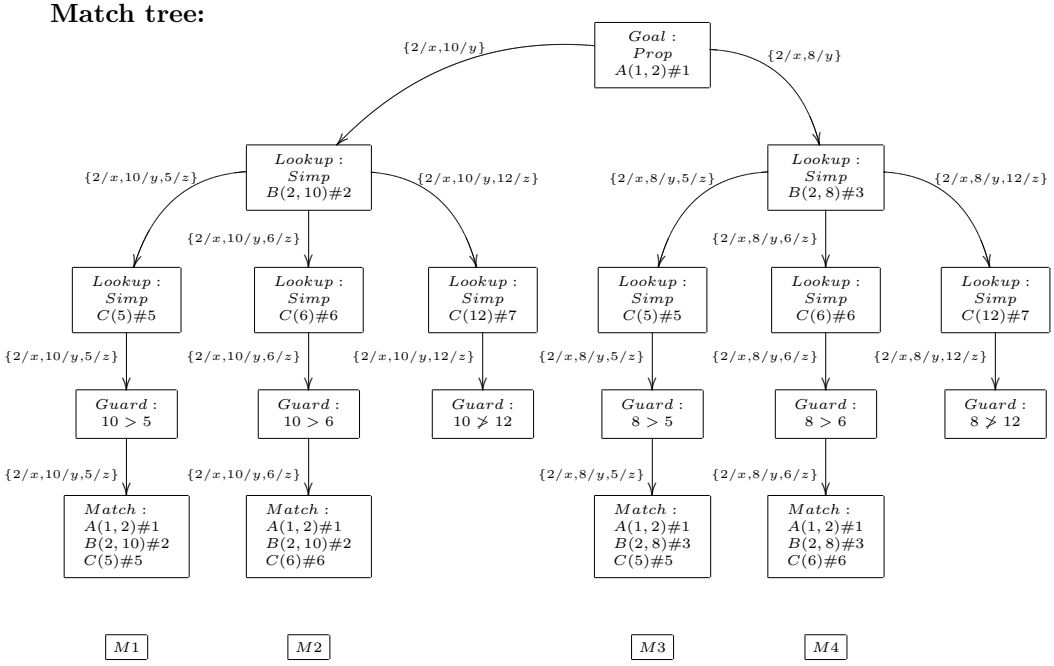


Figure 5.9: Example of CHR rule, derivation and match Tree

rule $r1$, we propagate $A(1,x)$ but simplify $B(x,y)$ and $C(z)$. If we choose to use match $M1$, match $M2$ becomes invalid because $M1$ and $M2$ share an overlapping constraint $B(2,10)\#2$ which will be simplified. Hence, we can either use match $M1$ or $M2$ but not both. The CHR semantics (eg. $\| \mathcal{G}$) of course does not any impose restriction on the choice of which match to use. Similarly, match $M3$ becomes invalid because of the shared simplified constraint $C(5)\#5$. Hence, for each match tree, we can only fire a set of rule head instances which has mutually non-overlapping simplified constraints. For instance, the following illustrates the $\| \mathcal{G}$ derivations that corresponds to the applications of matches $M1$ and $M4$.

$$\begin{array}{ll} & \langle \{A(1,2)\#1\} \mid \{A(1,2)\#1, B(2,10)\#2, B(2,8)\#3, C(5)\#4, C(6)\#5, C(12)\#6\} \rangle \\ \rightarrowtail_r & \langle \{A(1,2)\#1, D(2,10,5)\} \mid \{A(1,2)\#1, B(2,8)\#3, C(6)\#5\} \rangle \\ \rightarrowtail_r & \langle \{A(1,2)\#1, D(2,10,5), D(2,8,6)\} \mid \{A(1,2)\#1, C(12)\#6\} \rangle \end{array}$$

Similarly, we can apply the alternative set of matches $M2$ and $M3$. In general, we can apply any subsets of matches of a match tree which consist of mutually non-overlapping rule head matches.

Figure 5.1 illustrates pseudo code which implements the execution of goals of the form $A(1,x)\#n$. Description of operations `match`, `deleteFromStore` and `addToGoals` can be found in [Lam, 2010]. Line 2 creates an iteration ($ms1$) of constraints in the store Sn that matches the pattern $B(x,\_)$, where the $\_$ symbol represents the 'any' pattern. The 'For' loop of lines $3-13$ tries matching constraints in $ms1$ on the rest of the search procedure. Similar to Line 2, Line 4 creates an iteration of constraints matching $C(\_)$.

```
1  execGoal ⟨G | Sn⟩ A(1,x)#n {
2       ms1 = match Sn B(x,_)
3       for B(x,y)#m in ms1 {
4            ms2 = match Sn C(_)
5            for C(z)#p in ms2 {
6                 if(y > z) {
7                      deleteFromStore Sn [B(x,y)#m,C(z)#p]
8                      addToGoals G [A(1,x)#n,D(x,y,z)]
9                      return true
10                }
11           }
12      }
13      return false
14 }
```

Table 5.1: Example of basic implementation of CHR goal-based rewritings

```
1  exec_Goal:
2      while ∃ goal
3         select goal G
4         if ∃ r@ P1,...,Pl \ S1,...,Sm ⟺ tg | C1,...,Cn ∈ P and
5            ∃ φ such that
6            St ≡ Stc ⊎ {φ(P1),...,φ(Pl),φ(S1),...,φ(Sm)} and
7            ⊨ φ(tg) and
8            either (G ≡ φ(Pi) for some i ∈ {1,...,l}) or
9                   (G ≡ φ(Sj) for some j ∈ {1,...,m})
10        then let ψ be m.g.u. of all equations in C1,...,Cn
11             St := Stc ⊎ { φ(P1),...,φ(Pl), φ∘ψ(C1),...,φ∘ψ(Cn)}
```

Table 5.2: Goal-based lazy match rewrite algorithm for ground CHR

This is following by the inner 'For' loop of Lines $7-11$ which iterates through constraints in $ms2$. Line 6 checks the rule guard which only executes rewriting (Lines $7-9$) for constraint sets satisfying $y > z$. CHR rewriting is modeled by the following: Line 7 removes the constraints $B(x,y)\#m$ and $C(z)\#p$ which matched the simplified heads of the rule. Line 8 adds the rule body $D(x,y,z)$ and the propagated goal constraint $A(1,x)\#n$ [4] into the CHR goals G as new goal(s) to be executed later. Line 9 exits the procedure with success (true). Finally, in Line 13, when no rule head match is found, the goal constraint is dropped and the procedure is exited with failure (false). Note that this procedure essentially traverses the search space specified by match tree in Figure 5.9.

Existing implementations assume that goal execution routines such as the one found in Figure 5.1 are executed strictly in isolation , hence avoiding the issues and woes of concurrent execution. For the rest of the Chapter, we will detail these issues and highlight our solutions to address them.

Table 5.2 lays out the general structure of a goal-based lazy match rewrite algorithm. We select a goal G which then finds its matching partners. Lines 8 and 9 ensure that the goal must be part of the left-hand side. Our formulation assumes that the CHR rule system is *ground*. That is, equations on right-hand side of rules can immediately

[4] Note that this necessary, as specified by the (Propagate) rule of the $\| \mathcal{G}$ semantics.

be eliminated by applying the m.g.u. This ensures that any derivation starting from a ground constraint store (none of the constraints contains any free variables) can only lead to another ground constraint store. In our experience, the restriction to ground CHR is not onerous because most examples either satisfy this condition, or it is fairly straightforward to program unification/instantiation on top of CHR (e.g. see our encoding of union-find in the upcoming Section 5.6).

In essence, we wish to extend this CHR execution scheme to execute multiple copies of CHR rewritings (Table 5.2) concurrently, each copy strictly executing a distinct goal but rewriting over the same store St shared among all computation threads.

5.5 Parallel CHR System in Haskell GHC

In this section, we dive down into the details of implementing a concrete parallel CHR system, known as ParallelCHR , that implements the $\| \mathcal{G}$ semantics in a scalable manner. Our choice of programming language is Haskell, a lazy functional programming language. In particular, we use the Glasgow Haskell Compiler [GHC] because of its good support for shared memory, multi-core architectures. Haskell also provides high-level abstraction facilities (polymorphic types higher-order functions etc) and its clean separation between pure and impure computations invaluable in the development of our system. In principle, our system can be re-implemented in other main-stream languages such as C and Java. Our implementation in Haskell GHC is available for download at `http://code.google.com/p/parallel-chr/`.

5.5.1 Implementation Overview

As the most computationally intensive routine of CHR goal execution is the search for matching constraints, much can be gained by implementing a CHR system which can execute search routines (for matching constraints) of multiple CHR goals in parallel, over a shared constraint store. While the $\| \mathcal{G}$ semantics formally describes how CHR goals can be executed concurrently over a shared constraint store, it provides little details on how we can implement this in a practical and scalable manner. In other words, the technical concerns of how to implement scalable CHR rewritings are not observable in the formal semantics.

We take a high-level look at finding matches in parallel and atomic rule execution. In our implementation, a thread pool consisting of several light-weight Haskell GHC threads are used to execute CHR goals in a shared collection of goals. Goal execution threads like these execute CHR rewriting asynchronously by searching the shared store for matching partner constraints (to complete rule head instances), deleting the simplified constraints of the rule head instance and finally adding body constraints into the collection of goals. The challenge we face in this parallel execution problem is that the partners found by asynchronous threads running in parallel be overlapping (share similar simplified heads). As defined in the $\| \mathcal{G}$ semantics (Definition 5.2.1), parallel goal execution must rewrite over non-overlapping rule-heads. Here, we briefly introduce two approaches which uses different concurrency primitives to implement this non-overlapping parallel rule-head matching routine.

Fine-grained Lock-based parallel matching: This approach is a standard refinement of the coarse-grained locking approach. Rather than guarding the shared store with a single global lock, we restrict the access of each constraint in the shared store with a unique dedicated lock. The parallel matching task at hand now includes incrementally

acquire locks of partner constraints. However, we must be careful to avoid deadlocks. For example, suppose that thread 1 and 2 seek partners A and B to fire any of the rules $A, B, C \Longleftrightarrow rhs_1$ and $A, B, D \Longleftrightarrow rhs_2$. We assume that C is thread 1's goal constraint and D is the goal constraint of thread 2. Suppose that thread 1 first encounters A and locks this constraint. By chance, thread 2 finds B and imposes his lock on B. But then none of the two threads can proceed because thread 1 waits for thread 2 to release the lock imposed on B and thread 2 waits for the release of the locked constraint A.

The scenario illustrated above is a classic (deadlock) problem when programming with locks. The recently popular becoming concept of Software Transactional Memory (STM) is meant to avoid such issues. Instead of using locks directly, the programmer declares that certain program regions are executed atomically. The idea is that atomic program regions are executed optimistically. That is, any read/write operations performed by the program are recorded locally and will only be made visible at the end of the program. Before making the changes visible, the underlying STM protocol will check for read/write conflicts with other atomically executed program regions. If there are conflicts, the STM protocol will then (usually randomly) commit one of the atomic regions and rollback the other conflicting regions. Committing means that the programs updates become globally visible. Rollback means that we restart the program. The upshot is that the optimistic form of program execution by STM avoids the typical form of deadlocks caused by locks. In our setting, we can protect critical regions via STM as follows.

STM-based parallel matching means that we perform the search for partner constraints and their removal from the store atomically. For the above example, where both threads attempt to remove constraints A and B as well as their specific goal constraints we find that only one of the threads will commit whereas the other has to rollback, i.e. restart the search for partners.

The downside of STM is that unnecessary rollbacks can happen due to the conservative conflict resolution strategy. Here is an example to explain this point. Suppose that thread 1 seeks partner A and thread 2 seeks partner B. There is clearly no conflict. However, during the search for A, thread 1 reads B as well. This can happen in case we perform a linear search and no constraint indexing is possible or the hash-table has many conflicts. Suppose that thread 2 commits first and removes B from the store. The problem is that thread 1 is forced to rollback because there is a read/write conflict. The read constraint B is not present anymore. STM does not know that this constraint is irrelevant for thread 1 and therefore conservatively forces thread 1 to rollback.

In our current implementation, we use a **hybrid STM-based** scheme which uses both Software Transactional Memory and traditional shared memory access techniques. The search for matching partner constraints is performed "outside" STM (to avoid unnecessary rollbacks), this means that accessing constraint memory locations at this stage does not invoke STM concurrency synchronization protocols. Once a set of constraints forming a complete match is found, we perform an atomic STM procedure which atomically checks that all the constraints are still available, and logically deletes the simplified constraints[5]. This essentially implements *atomic rule-head verification* (as described in [Lam, 2010]) which guarantees the atomic deletion of rule-head instances. Logically deleted constraints will eventually be physically delinked from the constraint store, either immediately after the atomic rule-head verification step or by an amortized delete procedure. Both can be implemented with relative ease with traditional concurrency primitives (e.g. compare-and-swap, locks, etc...).

[5]By logically delete, we mean that the constraint is not physically removed from the data structure, but simply marked as deleted

Abstract Data Types

Integer Value:	`Int`	Boolean Value:	`Bool`
List of a's:	`[a]`	Substitution:	`Subst`
CHR Constraint:	`Cons`	Rule Guard:	`Guard`
CHR Store:	`Store`	CHR Goals:	`Goals`

Rule Occurrence Data Types

```
Head Type:        data Head = Simp | Prop
Match Task:       data MatchTask = LpHead Head Cons | SchdGrd Guard
Rule Compilation: type Comp = ([MatchTask],[Cons])
CHR Program:      type Prog = [Comp]
```

Figure 5.10: Interfaces of CHR data types

5.5.2 Data Representation and Sub-routines

We briefly discuss our data representation of the Constraint Handling Rules language in Haskell, illustrated by Figure 5.10. **Abstract Data Type** shows the Haskell data type representation of CHR language elements, like constraints, substitution, store etc. **Rule Occurrence Data Types** represent the goal-based compilation of CHR rules, detailed in Section 5.4.1. Essentially, a CHR Program is a list of CHR rule compilations. A rule compilation `Comp` is a tuple, which consist of a list of match tasks (`MatchTask`) and a list of constraints (`Cons`). Note that we will represent sets with lists. Note that for presentation, we shall focus entirely on CHR matching and rewriting of CHR constraints. Hence we will not include builtin constraints in our CHR language here. Treatment of builtin constraints can be found in [Lam, 2010].

The following provide brief descriptions of the basic CHR Solver Sub-routines. These sub-routines represents basic interfaces to the underlying shared store and goal data structures, as well as substitution framework.

- `isAlive :: Cons -> Bool`
  Given CHR constraint `c`, returns true if and only if `c` is still stored.

- `match :: Subst -> Cons -> Cons -> IO (Maybe Subst)`
  Given a substitution and two CHR constraints `c` and `c'`, returns resultant substitution of matching `c` with `c'`, if they match. Otherwise return nothing.

- `consApply :: Subst -> [Cons] -> [Cons]`
  Given a substitution and a list of CHR constraints, apply the substitution on each constraint of the list and return the results.

- `grdApply :: Subst -> Guard -> Bool`
  Given a substitution and a guard condition, apply the substitution on the guard and return true iff guard condition is satisfiable.

- `emptySub :: Subst`
  Returns the empty substitution.

- `addToStore :: Store -> Cons -> IO Cons`
  Given a CHR store `st` and a CHR constraint `c`, add `c` into `st`. Returns the stored

constraint c containing additional book-keeping information (store back-pointers, etc..).

- `getCandidates :: Store -> Cons -> IO [Cons]`
  Given a CHR Store `st` and a CHR constraint `c`, return a list of constraints from the `st` that matches `c`.

- `getGoal :: Goals -> IO (Maybe Cons)`
  Given CHR goals, returns the next goal if one exists, otherwise returns nothing.

- `addGoals :: Goals -> [Cons] -> IO ()`
  Given CHR goals `gs` and a list of CHR constraints `cs`, add `cs` into `gs`.

- `notRepeat :: [(Head,Cons)] -> Cons -> Bool`
  Given a list of matching heads, and a constraint **c** returns true if **c** is not already found in the list of heads.

- `isStored :: Store -> Cons -> STM Bool`
  Given a CHR store `st` and a constraint `c`, returns true if and only if `c` is stored in `st`.

- `logicalDeleteFromStore :: Store -> Cons -> STM ()`
  Given a CHR store and a constraint in the store, logically mark the specified constraint as deleted from the store.

- `delinkFromStore :: Store -> Cons -> IO ()`
  Given a CHR store and a constraint in the store, physically delink the constraint from the store.

- `atomically :: STM a -> IO a`
  Given a `STM` operation, execute it atomically in the `IO` monad.

Next, Section 5.5.3 and 5.5.4 will introduce the main high-level goal execution routine which uses these sub-routines.

5.5.3 Implementing Parallel CHR Goal Execution

We introduce our parallel CHR implementation from a top-down approach, starting from the function `goalBasedThread`, as shown in Table 5.3. The parallel CHR solver comprises of multiple copies of this function, executed asynchronously in parallel by multiple threads of computation. Each execution essentially implements the execution of a CHR goal. For now, we focus on execution of CHR goals only.

This function is given the references to the shared goals `gs` and store `st`, and the CHR program `prog`. Goals are exhaustively executed via the `rewritingLoop` procedure, which terminates only when the goals are empty (line 11). As specified by the (Activate) rule of the $\| \mathcal{G}$ semantics (Figure 5.4 of Section 5.2), goals are added to the store only when they are executed (line 8). Procedure `executeGoal` attempts to match the active goal `g` with each of the occurrence compilations (via the `matchGoal` operation at line 13, whose definition is deferred till later). Procedure `executeGoal` stops when the goal is no longer alive (line 15) or all occurrence have been tried (line 16), both of which are cases which lead to the application of the (Drop) rule of the $\| \mathcal{G}$ semantics.

```
1  goalBasedThread :: Goals -> Store -> Prog -> IO ()
2  goalBasedThread gs st prog =
3     rewriteLoop
4     where
5       rewriteLoop = do
6         { mb <- getGoal gs
7         ; case mb of
8            Just g -> do { a <- addToStore st g
9                         ; executeGoal a prog
10                        ; rewriteLoop }
11           Nothing -> return () }
12      executeGoal a (occ:occs) = do
13        { matchGoal gs st a occ
14        ; if isAlive a then executeGoal a occs
15                       else return () }
16      executeGoal _ [] = return ()
```

Table 5.3: Top-level CHR Goal Execution Routine

Procedure `matchGoal` in Table 5.4 implements the main parallel matching algorithm which searches for matching constraints of the active goal constraint. This search is specified by the match tasks of CHR goal-based rule compilations, described earlier in Section 5.4.1. We assume that the first match task is the lookup of the active goal pattern (line 3)[6]. In line 4 the active goal is then matched with the head pattern (from the lookup task)[7]. If the active goal successfully matches the head pattern (line 6) we call `execMatch`. If matching fails, we abort the procedure (line 7).

Procedure `execMatch` essentially implements the search traversal through CHR match trees (Section 5.4.2). It checks for the remaining match tasks, which can be either looking up a partner constraint, or checking a guard condition. It controls the branching of the search by returning $True$ if the search is to terminate at the current branch, or $False$ if the search is to proceed. Lines $9-11$ implements the scheduling of a guard constraint `grd`. We proceed with the rest of the match tasks if the guard evaluates to true. Lines $12-25$ on the other hand, implements the lookup of a partner constraint, as specified by the matchtask `LpHead hd c`. We first collect all possible candidates `cans` matching `c` from the store (line 24)[8]. Then, we call `exec_match_candidate` (line 25) which tries to find a complete match for the entire rule head by iterating over the set of candidates. Note that we only iterate through as many candidate as required (lines $19-20$) for exhaustiveness of the goal execution (details in Section 5.5.8).

In case we find a complete match (line 26), we fire the rule. Note that this step can happen in parallel with multiple goal executions, hence to guarantee consistency, we must atomically verify and commit this match via `verifyRuleHeads` (line 27). This procedure checks that all heads are still alive and logically marks all the simplified heads as deleted. All these operation are done in one atomic transactional step. That is, if any of the intermediate steps fails the entire transaction fails with no visible side effect (atomic rule-head verification, cf. Sect. 5.5.4). Follow a successful run of `verifyRuleHeads` (line $28-32$), we will physically delink all the simplified constraints (line 29) and add the body constraints of the rule instance into the goals (line 30). If the executed goal is a simplified

[6]This is because CHR rules have at least one head, hence this constraint lookup task must exist.

[7]Note that with known pre-compilation analysis, this matching of active goal and head pattern can be avoided. Such optimizations are covered in [Duck, 2005] and will not be discussed here.

[8]Note that this procedure can be implemented 'lazily', or with iterators representing a collection of all candidate matching constraints and hence only retrieved on demand.

```
1  matchGoal :: Goals -> Store -> Cons -> Occ -> IO ()
2  matchGoal goals store g (mtasks,body) = do
3   { let (LpHead hd c):rest = mtasks
4   ; mb <- match emptySub c g
5   ; case mb of
6      Just subst -> do { execMatch [(hd,g)] subst rest ; return () }
7      Nothing    -> return () }
8   where
9    execMatch hds subst ((SchdGrd guard):mts) =
10    if grdApply subst guard then execMatch hds subst mts
11                            else return False
12   execMatch hds subst ((LpHead hd c):mts) =
13    let execMatchCandidates (nc:ncs) =
14         if (notRepeat hds nc) && (isAlive nc)
15         then do { mb <- match subst c nc
16                 ; case mb of
17                    Just subst' -> do
18                        { succ <- execMatch ((h,nc):hds) subst' mts
19                        ; if not succ then execMatchCandidates ncs
20                                      else return False }
21                    Nothing     -> execMatchCandidates ncs }
22         else execMatchCandidates ncs
23        execMatchCandidates [ ] = return False
24    in do { cans <- getCandidates store c
25          ; execMatchCandidate cans }
26   execMatch hds subst [ ] = do
27    { succ <- atomically (verifyRuleHeads store hds)
28    ; if succ then do { let simpHds = filter (\(h,_) -> h == Simp) hds
29                      ; mapM (\(_,g) -> delinkFromStore store g) simpHds)
30                      ; addGoals goals (consApply subst body)
31                      ; let (h,_) = first hds
32                      ; return (h == Simp) }
33             else return False }
```

Table 5.4: Implementation of Goal Matching

head, we end the search by return *True* (since the goal currently executed will be deleted from the store), otherwise we proceed to the next candidate (line 32). In a failed run of `verifyRuleHeads` (line 33), we return *False* to indicate that the goal execution should try another partner constraint.

Note that the delinkings of simplified constraints (line 29) are done in a seemingly unsafe ("unatomic") sequence of `IO` operations. Yet it is safe to do so, thanks to the fact that the constraints to be delinked at line 29 are the same constraints marked as deleted by `verifyRuleHeads` in line 27. Hence we have the guarantee that no two concurrent goal executions will attempt to delink the same constraints.

5.5.4 Implementing Atomic Rule-Head Verification

Atomic rule-head verification (ARV) guarantees the atomic deletion of rule-head instances. We detail here the implementation via Software Transactional Memory in Haskell (GHC).

Table 5.5 illustrates the implementation of ARV with STM in Haskell GHC. The STM operation `verifyRuleHeads` works as follows: Given the shared store and a set of matching constraints (presumably the complete rule-head instance), we check that all the constraints are still in the store and not deleted by any other parallel goal execution routines (line

```
1  verifyRuleHeads :: Store -> [(Head,Cons)] -> STM Bool
2  verifyRuleHeads store hds = do
3   { bs <- mapM (\(_,g) -> isStored store g) hds
4   ; if and bs
5     then do { let simpHds = filter (\(h,_) -> h == Simp) hds
6             ; mapM (\(_,g) -> logicalDeleteFromStore store g) simpHds
7             ; return True }
8     else return False }
```

Table 5.5: Implementation of Atomic Rule-Head Verification (ARV)

3). If so (line $5-7$) we delete (from the store) all the constraints that are matched as simplified heads and return true. Otherwise (line 8) we return false. Note that since this STM operation is guaranteed to execute atomically, a successful run (resulting to the return of `True`) indicates that we were able to independently observe the presence of all constraints involved in the store and delete the simplified constraint. Most importantly, constraints involved in this STM validation process corresponds directly to the constraints that form the rule head instance, thus we will not introduce any false overlaps, which are described in the following section.

5.5.5 False Overlaps

When implementing concurrent CHR rewritings, it is no doubt that consistency of such an implementation is of the utmost importance. As such, we must exploit the concurrency synchronization protocols offered by most main-stream programming language, to model the consistent concurrent behaviours of the concurrent CHR semantics. Consider a simple concurrent derivation:

$$get@G(x), P(y) \Longleftrightarrow x = y$$

$$\begin{array}{ll}
 & \langle \{P(1)\#3, P(2)\#4\} \mid \{P(1)\#3, P(2)\#4, G(x)\#1, G(y)\#2\}\rangle \\
\overset{\{\}\backslash\delta_1}{\rightarrowtail^*_{||\mathcal{G}}} & \langle \{P(2)\#4\} \mid \{x = 1, P(2)\#4, G(y)\#2\}\rangle \qquad (1) \\
 & \qquad\qquad || \\
 & \langle \{P(1)\#3, P(2)\#4\} \mid \{P(1)\#3, P(2)\#4, G(x)\#1, G(y)\#2\}\rangle \\
\overset{\{\}\backslash\delta_2}{\rightarrowtail^*_{||\mathcal{G}}} & \langle \{P(1)\#3\} \mid \{y = 2, P(1)\#3, G(x)\#1\}\rangle \qquad (2) \\
 & \delta_1 = \{P(1), G(x)\} \qquad \delta_2 = \{P(2), G(y)\} \\
\hline
 & \langle \{P(1)\#3, P(2)\#4\} \mid \{P(1)\#3, P(2)\#4, G(x)\#1, G(y)\#2\}\rangle \\
\overset{\{\}\backslash(\delta_1 \uplus \delta_2)}{\rightarrowtail^*_{||\mathcal{G}}} & \langle \{\} \mid \{x = 1, y = 2\}\rangle
\end{array}$$

This concurrent derivation shows a valid non-overlapping pair of CHR derivations (labelled (1) and (2)) which can be concurrently composed together. They are non-overlapping because their side-effects $\{\}\backslash\{P(x), G(1)\}$ and $\{\}\backslash\{P(y), G(2)\}$ consist of mutually distinct sets of constraints. To only allow such non-overlapping concurrent derivations in an implementation of the concurrent semantics, we use concurrency synchronization protocols to model the enforcement of mutual exclusivity of CHR derivation side-effects. While it is not always possible (or practical) to exactly model such mutual exclusiveness of such side-effects, any implementation must guarantee that it models mutual exclusive side-effects conservatively, in that it *never allows overlapping parallel*[9] *deriva-*

[9] note we use parallel here as oppose to concurrent. This is because we are talking about actually

tions but *might deny certain non-overlapping parallel derivations* specifically because of false overlapping side-effects, introduced by the conservative concurrency synchronization protocols. An example of such false overlaps can be as follows: Suppose we implement our parallel CHR derivations by making the execution of a derivation demand complete "atomicity", in that all shared memory (in this case, constraints in the store) that it has read and written to must not have been modified by another parallel execution, otherwise it must retry it's execution. Further suppose that our store is naively implemented with a simple concurrent linked-list and all CHR execution threads search the store in a specific order (for convenience, let's assume left-to-right ordering for the above example). Execution of (1) would have read constraints $\{P(1)\#3, P(2)\#4, G(x)\#1\}$ before finding the rule head match $\{P(1)\#3, G(x)\#1\}$ while execution of (2) would have read $\{P(1)\#3, P(2)\#4, G(x)\#1, G(y)\#2\}$ before finding $\{P(2)\#4, G(y)\#2\}$. As such, the concurrent derivations would be ruled as overlapping, and cannot be allowed to execute in parallel.

Such false overlaps, caused by the in-precision of coarse grained concurrency synchronization, normally impairs the benefits of parallel execution of CHR rewritings. As a result, a practical and scalable implementation of parallel CHR rewriting must be implemented to model exclusivity of concurrent CHR derivation side-effects conservatively, while minimizing false overlaps as much as possible.

5.5.6 Logical Deletes and Physical Delink

Recall that we have chosen to use logical deletes during ARV, while the physical delinking of constraints from the store data structure is only executed in subsequent non-atomic steps (Sections 5.5.3 and 5.5.4). This approach is beneficial in two ways. Firstly, we can implement ARV with **smaller STM transactions**. This is because multiset logical deletes can be straight-forwardly implemented as the toggling of boolean flags stored in STM transactional variables. Hence, logically deleting n constraints is essentially just writing into n boolean variables. Logical deletes are much cheaper operations, compared to implementing physical removal of constraints (from the store) which involves delinking of nodes from a list data structure (implemented on STM). As such, our atomic rule-head verification can be implemented with smaller STM transactions which most certainly incur less conflicts from STM roll backs. Besides reducing the number of STM roll backs, we can now implement other list operations (list traversal, delinking of list nodes) via **lighter weight concurrency primitives**.

In our works on comparing Haskell concurrency primitives [Sulzmann et al., 2008], we have demonstrated with empirical evidence that a concurrent list data structure implemented via traditional compare-and-swap operations is much more efficient than one implemented via STM. Yet, STM provides the most elegant solution to atomic multiset operations[10]. Our multiset logical delete via STM and physical delink via compare-and-swap implementation essentially adopts the best of both worlds (or rather, concurrency primitives) and provides the alternative with least concurrency synchronization overheads.

5.5.7 Back Jumping in ARV

Consider the following example:

$$A(x), B(x, y), C(y, z), D(z) \Longleftrightarrow E(x, y, z)$$

executing CHR derivations concurrently, in practice.

[10] as oppose to traditional locks or compare-and-swap synchronization variables, which are prone to errors and other overheads incurred by complex synchronization acrobatics.

```
1  verifyRuleHeadsBackJump :: Store -> [(Head,Cons)] -> STM Int
2  verifyRuleHeadsBackJump store hds = do
3   { bs <- mapM (\(_,g) -> isStored store g) hds
4   ; if and bs
5     then do { let simpHds = filter (\(h,_) -> h == Simp) hds
6             ; mapM (\(_,g) -> logicalDeleteFromStore store g) simpHds
7             ; return 0 }
8     else let Just j_index = elemIndex False bs
9          in return (j_index + 1) }
```

Table 5.6: ARV with Backjumping Indicator

Suppose while executing the goal $A(1)\#n$ we have found the rule-head instance $[A(1)\#n, B(1,2)\#m, C(2,3)\#p, D(3)\#q]$ in that specific sequence and now attempts to rule ARV on the four constraints. Further suppose that the verification procedure failed because the constraint $B(1,2)\#m$ has already been deleted by some other executing thread. Our implementation of ARV in Table 5.5 will return $False$ suggesting that one of the constraint has been deleted, but without more information other than the boolean flag, our goal execution procedure in Table 5.4 has to explore other alternate branches of the match tree, iterating through possible alternative candidates of $D(z)$, $C(y,z)$, before reaching $B(x,y)$ lookup node, where the verification had failed.

To avoid such pointless traversals of the match tree, we can implement a well known optimization technique for backtracking search algorithms, known as **backjumping**. By keeping track of exactly which constraint has failed the ARV, we can precisely backtrack our search to the "highest" point of the match-tree which is possibly still valid and resume the search from that point.

Table 5.6 illustrates the ARV function `verifyRuleHeads` with backjumping indicator. `verifyRuleHeadsBackJump` is similar to that of in Table 5.5, but instead returns an integer. If verification is successful, 0 is returned (line 7). Otherwise, it returns the 1-index of the left-most constraint which has failed the verification[11]

Table 5.7 illustrates the modified `matchGoalBackJump` operation that utilizes the backjumping indexes provided by `verifyRuleHeadsBackJump`. Note that the most important change is in lines $19-20$, where new candidates are tried only if the jump index i returned by the previous branch (line 18) is equal to 1. Otherwise, we simply return the index decrement by one (line 20). Successful run of the `verifyRuleHeadsBackJump` indicated by index $i == 0$ (line 28) results to the same delinking of simplified constraint (line 29) and adding of body constraints into the goals (line 30). If goal is a simplified constraint (line 32), we return the number of the rule heads (effective "backjumping out" of the goal execution), otherwise we procedure on with the search through the match tree.

5.5.8 Implementation and $\| \mathcal{G}$ Semantics

In this section, we informally discuss the correspondence of our parallel CHR system in Haskell GHC, and the $\| \mathcal{G}$ semantics. Our implementation implements the CHR $\| \mathcal{G}$ semantics in that given the shared goals gs (initially containing goals cs), shared store st (initially empty) and CHR program compilation $prog$ (of CHR program $\mathcal{P}$), when multiple concurrent execution of the `goalBasedThread gs st prog` goal execution routine terminates[12], shared goals gs will be empty and shared store st will contain constraints

[11] Since new rule-heads are appended to the end of our rule-heads hds, left-most constraint which has failed represents the "highest" point of the match tree which has failed the verification.

[12] This also includes the termination of all goal reactivation threads

```
1  matchGoalBackJump :: Goals -> Store -> Cons -> Occ -> IO ()
2  matchGoalBackJump goals store g (mtasks,body) = do
3   { let (LpHead hd c):rest = mtasks
4   ; mb <- match emptySub c g
5   ; case mb of
6      Just subst -> do { execMatch [(hd,g)] subst rest ; return () }
7      Nothing    -> return () }
8   where
9    execMatch hds subst ((SchdGrd guard):mts) =
10    if grdApply subst guard then execMatch hds subst mts
11                            else return 1
12   execMatch hds subst ((LpHead hd c):mts) =
13    let execMatchCandidates (nc:ncs) =
14         if (notRepeat hds nc) && (isAlive nc)
15         then do { mb <- match subst c nc
16                 ; case mb of
17                    Just subst' -> do
18                        { i <- execMatch ((h,nc):hds) subst' mts
19                        ; if i == 1 then execMatchCandidates ncs
20                                    else return (i-1) }
21                    Nothing     -> execMatchCandidates ncs }
22         else execMatchCandidates ncs
23        execMatchCandidates [ ] = return 1
24    in do { cans <- getCandidates store c
25          ; execMatchCandidate cans }
26   execMatch hds subst [ ] = do
27    { i <- atomically (verifyRuleHeadsBackJump store hds)
28    ; if i==0 then do { let simpHds = filter (\(h,_) -> h == Simp) hds
29                      ; mapM (\(_,g) -> delinkFromStore store g) simpHds)
30                      ; addGoals goals (consApply subst body)
31                      ; let (h,_) = first hds
32                      ; return (if h == Simp then (length hds) + 1 else 1) }
33              else return i }
```

Table 5.7: Goal Matching with Back-Jumping

cs' such that for the CHR program $\mathcal{P}$, $\langle cs \mid \{\}\rangle \rightarrowtail_{\|\mathcal{G}} \langle\{\} \mid cs'\rangle$.

To summarize, each $\| \mathcal{G}$ transition rule (Figure 5.4) corresponds to our implementation in the following manner:

- (Solve): An equation constraint e is not physically stored in the constraint st as suggested in the $\| \mathcal{G}$ semantics, but imposes its side-effects on the builtin theory when e is executed by the routine `solve`. The set of constraints $WakeUp(e, st)$ awaken (reactivated) by the (Solve) transition is replicated in our implementation by the execution of `reactivateWhenGround` by reactivation threads spawned specifically for this purpose (add the affected constraint back into the goals).

- (Activate): This transition immediately corresponds to the execution of `addToStore`, line 8 of `goalBasedThread` in Table 5.3.

- (Simplify): This transition corresponds to an execution of `executeGoal a prog` (line 9, Table 5.3) which results to the deletion of active goal constraint `a` (line 29, Table 5.4). This means that the active constraint `a` is one of the simplified constraint of the successful rule-head match. The CHR rewriting (removal of simplified constraints

and adding of body to the goals) is eventually completed by the execution of lines 28 − 32 of Table 5.4. Atomic rule head verification (line 27, Table 5.4) guarantees that concurrent goal execution selects mutually exclusive simplified constraints, hence rewrites non-overlapping rule instances. This is exactly specified by the merging of side-effects δ in the $\| \mathcal{G}$ semantics (Figure 5.5).

- (Propagate): Very much similar to the above (Simplify), except the execution of `executeGoal a prog` results in a successful rule-head match where `a` matches a propagated constraint.

- (Drop): This transition models the removal of a goal constraint after it has exhaustively searched the store for matching partner constraints and has not been simplified. It corresponds to the end of an execution of `executeGoal a prog` which does not end with the simplification of the active constraint `a`. Essentially, all complete execution of `executeGoal a prog` corresponds to a either a sequence (empty allowed) of (Propagate) transitions followed by a (Drop) transition, or a sequence (empty allowed) or (Propagate) transitions followed by a (Simplify) transition.

Our parallel CHR implementation faithfully implements the concurrent CHR goal-based semantics, in that every execution on a termination CHR program corresponds to a valid $\| \mathcal{G}$ concurrent derivation. But because of practical limitations of hardware, it is likely that our implementation cannot replicate all possible executions modeled by the semantics.

5.6 Experimental Results

In this section, we present the experiments we have conducted on our parallel CHR system and the empirical results we have collected. We focus on eight distinct CHR programs, which represents a diverse spread of CHR rules with varying characteristics. The following highlights each of these CHR programs, and the experiment parameters we have used:

- **Merge Sort**:

$$\begin{aligned} &\mathit{merge1} \ @ \ \mathit{Next}(x,a) \setminus \mathit{Next}(x,b) \iff a < b \mid \mathit{Next}(a,b) \\ &\mathit{merge2} \ @ \ \mathit{Merge}(n,a), \mathit{Merge}(n,b) \iff a < b \mid \mathit{Next}(a,b), \mathit{Merge}(n+1,a) \end{aligned}$$

  CHR implementation of Merge sort. CHR goal threads essentially compare different pairs of integers in parallel. We optimize with a specific goal ordering scheme (stack $Next$ goals and queue $Merge$ goals) which minimizes the number of comparisons between $Next$ constraints and the number of conflicts between goal executions (see [Lam, 2010] for details). For our experiment, we run merge sort on a collection of 1024 integers.

- **Gcd**:

$$\begin{aligned} &\mathit{gcd1} \ @ \ \mathit{Gcd}(n) \setminus \mathit{Gcd}(m) \iff m \geq n \&\& n > 0 \mid \mathit{Gcd}(m-n) \\ &\mathit{gcd2} \ @ \ \mathit{Gcd}(0) \iff \mathit{True} \end{aligned}$$

  CHR implementation of greatest common divisor Euclid's algorithm. We optimize by queuing Gcd goals. For our experiments, we find the greatest common divisor of 1000 integers. Finding the Gcds of distinct pairs of integers can be executed in parallel.

- **Parallelized Union Find**:

$$union \ @ \ Union(a,b), Fresh(x) \iff Fresh(x+2), Find(a,x), Find(b,x+1), Link(x,x+1)$$
$$findNode \ @ \ Edge(a,b) \setminus Find(a,x) \iff Find(b,x)$$
$$findRoot \ @ \ Root(a) \setminus Find(a,x) \iff Found(a,x)$$
$$found \ @ \ Edge(a,b) \setminus Found(a,x) \iff Found(b,x)$$
$$linkeq \ @ \ Link(x,y), Found(a,x), Found(a,y) \iff True$$
$$link \ @ \ Link(x,y), Found(a,x), Found(b,y), Root(a), Root(b) \iff Edge(b,a), Root(a)$$

Adapted from [Frühwirth, 2005], Union find is basically a data structure which maintains the union relationship among disjoint sets. Sets are represented by trees ($Edge(x,y)$) in which root notes ($Root(x)$) are the representatives of the sets. The union operation between two sets of a and b ($Union(a,b)$) is executed by finding the representatives x and y of the sets a and b ($Find(a,x)$ and $Find(b,y)$), and then linking them together ($Link(x,y)$). The *union* rule initiates the union operation. The constraint $Fresh(x)$ introduces "fresh variables" since our current prototype only supports ground CHR rules/stores. Rule *findNode* traverses edges until we reach the root in rule *foundRoot*. Rule *found* re-executes a find if the tree structure has changed. This is necessary since union find operations can be executed in parallel. Rule *linkeq* removes redundant link operations and rule *link* performs the actual linking of two distinct trees. In experiments, we test an instance of parallelized union find, where 300 union operations are issued in parallel to unite 301 disjoint sets (binary trees) of depth 5.

- **Blockworld**:

$$grab \ @ \ Grab(r,x), Empty(r), Clear(x), On(x,y) \iff Hold(r,x), Clear(y)$$
$$puton \ @ \ PutOn(r,y), Hold(r,x), Clear(y) \iff Empty(r), Clear(x), On(x,y)$$

A simple simulation of robot arms re-arranging stacks of blocks. $Grab(r,x)$ specifies that robot r grabs block x, only if r is empty and block x is clear on top and on y ($On(x,y)$). The result is that robot r will be holding block x ($Hold(r,x)$) and block x is no longer on block y, thus y is clear. $PutOn(r,y)$ specifies that robot r places a block on block y, if r is holding some block x and y is clear. In our experiments, we simulate 8 agents each moving a unique stack of 1000 blocks. Robots can be executed in parallel as long as their actions do not interfere.

- **Dining Philosophers**:

$$grabforks \ @ \ Think(c,0,x,y), Fork(x), Fork(y) \iff Eat(c,20,x,y)$$
$$thinking \ @ \ Think(c,n,x,y) \iff n > 0 \mid Think(c,n-1,x,y)$$
$$putforks1 \ @ \ Eat(0,0,x,y) \iff Fork(x), Fork(y)$$
$$putforks2 \ @ \ Eat(c,0,x,y) \iff Fork(x), Fork(y), Think(c-1,20,x,y)$$
$$eating \ @ \ Eat(c,n,x,y) \iff Eat(c,n-1,x,y)$$

The classic dining philosopher problem, simulating a group of philosophers thinking and eating on a round table, and sharing a fork with each of her neighbors. In our implementation, Forks are represented by the constraints $Fork(x)$ where x is a unique fork identifier. A thinking and eating philosopher is represented by the constraints $Think(c,n,x,y)$ and $Eat(c,n,x,y)$ where x and y are the fork identifiers, c represents the number of eat/think cycles left and n a counter that simulates the delay of thinking/eating process. Rules *thinking* and *easting* delay thinking and eating. If there any think/eat cycles left, we return both forks and issue a new thinking process. See rule *putforks2*. Otherwise, we only return both forks. See rule *putforks1*. In our experiments, we simulated the dining philosopher problem with 150 philosophers, each eating and thinking for 50 cycles with a delay of 20 steps.

- **Prime**:

$$\begin{gathered} \mathit{prime1} \ @ \ \mathit{Candidate}(1) \iff \mathit{True} \\ \mathit{prime2} \ @ \ \mathit{Candidate}(x) \iff x > 1 \mid \mathit{Prime}(x), \mathit{Candidate}(x-1) \\ \mathit{prime3} \ @ \ \mathit{Prime}(y) \setminus \mathit{Prime}(x) \iff x \bmod y == 0 \mid \mathit{True} \end{gathered}$$

  A CHR program that computes the first n prime numbers. In our experiments, we find the first 1500 prime numbers. Parallelism comes in the form of parallel comparison of distinct pairs of candidate numbers.

- **Fibonacci**:

$$\begin{gathered} \mathit{fibo1} \ @ \ \mathit{FindFibo}(0) \iff \mathit{Fibo}(1) \\ \mathit{fibo2} \ @ \ \mathit{FindFibo}(1) \iff \mathit{Fibo}(1) \\ \mathit{fibo3} \ @ \ \mathit{FindFibo}(x) \iff \mathit{FindFibo}(x-1), \mathit{FindFibo}(x-2) \\ \mathit{fibo4} \ @ \ \mathit{Fibo}(x), \mathit{Fibo}(y) \iff \mathit{Fibo}(x+y) \end{gathered}$$

  A CHR program that computes the value of the n^{th} Fibonacci number. We find the 25^{th} Fibonacci number. Parallelism is present when evaluating different parts of the Fibonacci tree.

- **Turing Machine:**

$$\begin{aligned} \mathit{delta\_left} \quad & @ \ \mathit{Delta}(qs, ts, qs', ts', \mathit{LEFT}) \setminus \mathit{CurrState}(i, qs), \mathit{TapePos}(i, ts) \\ & \iff \mathit{CurrState}(i-1, qs'), \mathit{TapePos}(i, ts') \\ \mathit{delta\_right} \quad & @ \ \mathit{Delta}(qs, ts, qs', ts', \mathit{RIGHT}) \setminus \mathit{CurrState}(i, qs), \mathit{TapePos}(i, ts) \\ & \iff \mathit{CurrState}(i+1, qs'), \mathit{TapePos}(i, ts') \end{aligned}$$

  A simple formulation of the classic Turing machine. In our implementation *delta_left* and *delta_right* define the state transitions of the Turing machine. The constraint $\mathit{Delta}(qs, ts, qs', ts', dir)$ specifies the state transition $(qs, ts) \mapsto (qs', ts', dir)$ where qs, qs' are state symbol and ts, ts' are tape symbols and dir is the direction which the tape is moved. $\mathit{CurrState}(i, qs)$ states that the current state of the machine is qs at tape position i. $\mathit{TapePos}(i, ts)$ states that tape position i has the symbol ts. In our experiments, we tested a Turing machine instance which determines if a tape (string of 0's and 1's) of length 200 is of the form $\{0^n 1^n \mid n > 1\}$. The Turing machine simulator is inherently single thread (rules cannot fire in parallel), as it involves state transitions of a single state machine. This serves to investigate the effects of parallel rewriting applied to a single threaded problem.

Our experiments are conducted to investigate into the effects of optimizations targeted at improving parallel goal execution. We observe the performance of the eight CHR programs with each optimization versus a default alternative. To summarized, we focus on the following concurrency specific optimizations:

- **Throttled/Bounded Thread Pools**: Aimed to reduce number of conflicting parallel executions and to prevent limited system resources from being swarmed by redundant concurrent goal executions. The alternative to this is to rely entirely on GHC's thread pooling system, hence we spawn a lightweight GHC thread to execute each new active goal.

- **Atomic Rule-Head Verification (ARV)**: Aimed to reduce the number of false-overlaps during parallel goal executions. The alternative to this is a simple STM implementation that does not use ARV (This implementation executes each goal as a single STM operation).

	1 Thread	2 Threads	4 Threads	8 Threads	Unbounded
Merge Sort	121%	94%	70%	52%	>200%
Gcd	109%	37%	18%	12%	123%
Parallel Unionfind	125%	82%	52%	32%	>200%
Blockworld	123%	77%	54%	39%	>200%
Dining Philosophers	119%	74%	49%	41%	>200%
Prime	115%	73%	46%	30%	155%
Fibonacci	125%	85%	59%	39%	>200%
Turing Machine	111%	63%	78%	70%	>200%

Figure 5.11: Experimental results, with optimal configuration (on 8 threaded Intel processor)

- **Bag Constraint Store and Store Iterators**: Aimed to reduce number of overlapping matches selected by parallel goals, by making each goal thread observe stored constraints in a unique order. The alternative to this are basic list constraint stores and list store iterators.
- **Domain Specific Goal Ordering**: Aimed to optimally schedule goals for execution. Goal ordering is specifically customized for each CHR program and only crucial for some examples, specifically *Gcd* and *Mergesort*. The alternative to this is the basic stack ordering of goals, which is the traditional ordering used by most CHR implementations.

On top of the concurrency optimizations mentioned here, our implementation also includes existing CHR optimizations which are still applicable to the concurrent context. Specifically, our implementation includes constraint indexing (hashing), optimal join ordering and early guard scheduling.

Our experiments are conducted on an Intel Core i7-920 processor[13] with 6 GB of memory running 64-bit Windows XP and Haskell GHC 6.10.1. For each experiment, we measure the relative performance of executing with 1, 2, 4 and 8 goal thread(s) against a *base* non-concurrent implementation in Haskell. Final results shown are the medians of 20 runs of the same experiment. This non-concurrent implementation serves as a benchmark for our concurrent implementation and is free from the overheads of concurrent execution (e.g. invoking STM runtime synchronization, ARV, etc..).

5.6.1 Results with Optimal Configuration

Figure 5.11 illustrates the experimental results conducted with our parallel CHR system in optimal configuration. In other words, **ARV**, **Bag constraint store and iterators**, **throttled goal thread pool** and **domain specific goal ordering**[14] concurrent optimizations are enabled. Measurements are based on the percentage time against execution time of the basic non-concurrent implementation (we will refer to this execution time as the base execution time).

There are several important observations. Firstly, executing our parallel implementation with 1 goal thread is inferior (at all examples) compared to the non-concurrent implementation for obvious reasons (overheads of concurrent execution are introduced,

[13] An Intel Core i7-920 processor is essentially a quad core processor, but is equipped with Hyper-threading technology that effectively allows it to run 8 concurrent threads of computation.

[14] Where applicable. Namely, Merge sort and Gcd

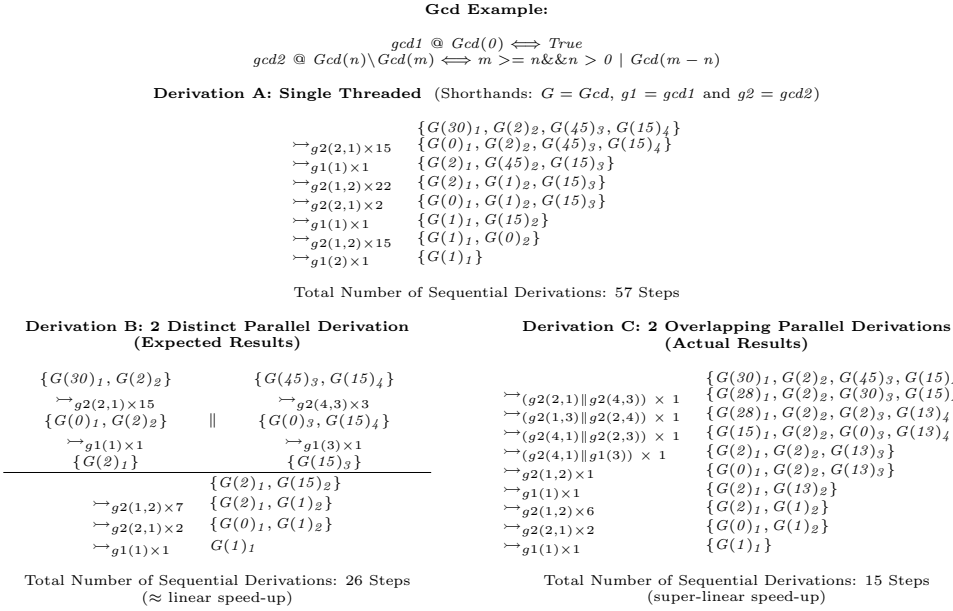


Figure 5.12: Super-Linear Speed-up in Gcd

with no benefits of concurrent goal execution being exploited). Executions with 2, 4 and 8 goal threads scale well against base execution time in general, with exception of the Turing Machine example. This is expected as the Turing machine example is inherently single-threaded. Interestingly, we still obtain improvements from parallel execution of administrative procedures (for example dropping of goals, due to failed matching). Relative drop in performance (between 2 and 4/8 goal threads) indicates a upper bound of parallelism of such "administrative" procedures.

One interesting result that our experiment uncovered is the presence of super-linear speed-up for certain examples, like Gcd. The reasons for this is often very subtle and domain specific. Figure 5.12 illustrates why we get super-linear speed-up for the Gcd example. For presentation purpose, we annotate each constraint with a unique identifier and each derivation with the rule name parameterized by the constraints that fired it and the number of times it fired. For instance $g2(x, y) \times t$ denotes that rule $g2$ fired on constraints x and y for t number of times. We examine derivations of the Gcd example from the initial store $\{Gcd(30), Gcd(2), Gcd(45), Gcd(15)\}$ Derivation A shows the single threaded case where we get a total of 57 derivation steps to reach the final store. Derivation B shows the parallel derivation of 2 threads which yield the expected results (linear speed-up of 26 sequential derivation steps). This assumes an unlikely scenario where derivations between 2 pairs of Gcd constraints do not overlap (i.e. interfere with each other). Derivation C shows the actual result which yields super-linear speed-up. Derivations overlap, that is, there can be rule firings across parallel derivations. [15] This allows Gcd constraints of higher values to be matched together, cutting down tediously long derivations initiated by

[15] Of course, this behavior is also possible in a sequential execution scheme where we interleave the execution of goal constraints, thus, effectively simulating the parallel execution scheme.

	1 Thread	2 Threads	4 Threads	8 Threads
Merge Sort	118%	143%	>200%	>200%
Gcd	123%	>200%	>200%	>200%
Parallel Unionfind	109%	131%	152%	178%
Blockworld	119%	126%	189%	>200%
Dining Philosophers	124%	89%	72%	75%
Prime	115%	119%	147%	175%
Fibonacci	124%	>200%	>200%	>200%
Turing Machine	111%	73%	82%	85%

Figure 5.13: Experimental results, with atomic rule-head verification disabled

Gcd constraints of lower values (which is typical in the single threaded case). By queuing *Gcd* goals (domain specific goal ordering), we encourage derivations similar to Derivation C to be chosen over other possibilities, since goals are processed in a breadth first manner (See results in Section 5.6.4 for confirmation of this point).

The final important insight lies in the right most data set of each CHR example. "Unbounded" refers to the performance of the parallel CHR system when we do not bound the number of CHR goal threads. This means that we spawn as many Haskell GHC lightweight threads as there are goal constraints, hence representing the abandonment of the **bounded goal thread pool** concurrency optimization. Results here show definitively that unbounded thread pooling (see [Lam, 2010]) is harmful to parallel CHR goal execution, with all CHR examples in this configuration performing sub-optimally.

5.6.2 Disabling Atomic Rule Head Verification

Figure 5.13 illustrates the experiment results conducted with **ARV** disabled. The alternative implementation we use here is similar to the simple implementation described in [Lam, 2010] and has the potential to introduce many false overlaps in concurrent goal execution. In general, results here show that multi-threaded goal execution performs worse than a single threaded execution or even the basic non-concurrent implementation. This essentially highlights the importance of minimizing false overlaps in concurrent goal execution, via ARV, or other fine-grained micro management of lower level concurrency primitives.

Dining Philosophers and Turing Machine examples demonstrate slight speed ups over base execution time, showing that there are domains which are more tolerant to the absence of fine-grained synchronization (introduced by ARV). It is not surprising, since Dining philosophers and Turing machine are examples in which CHR rule head matching heavily relies on constraint indexing. For instance, looking at the dining philosopher's problem, the active goal $Think(c, 0, a, b)\#n$ can seek for partners $Fork(a)\#m$ and $Fork(b)\#p$ via specifying indexed lookups for arguments a and b in the $Fork$ constraint store (as oppose to a linear iteration of all $Fork$ constraint, until a and b are found). This reduces the number of shared memory reads and thus reducing number of false overlaps, even without the presence of streamlined synchronization introduced by ARV.

To further support this argument, we investigate further by repeating the experiments, this time with constraint indexing also disabled. Figure 5.14 shows the highlights of this follow up experiment (we omit the examples which present insignificant or no difference from results in Figure 5.13). In this experiment, we see that Unionfind, Dining philosophers and Turing machine demonstrate terrible performance when constraint indexing is disabled. Since these examples heavily rely on constraint indexing, using linear lookups

	1 Thread	2 Threads	4 Threads	8 Threads
P. Unionfind (no ARV)	109%	131%	152%	178%
P. Unionfind (no ARV, no Indexing)	114%	>200%	>200%	>200%
Dining Philo (no ARV)	124%	89%	72%	75%
Dining Philo (no ARV, no Indexing)	126%	>200%	>200%	>200%
Turing Machine (no ARV)	111%	73%	82%	85%
Turing Machine (no ARV, no Indexing)	111%	>200%	>200%	>200%

Figure 5.14: Experimental results with and without constraint indexing (atomic rule-head verification disabled)

	1 Thread	2 Threads	4 Threads	8 Threads
Gcd (Optimal)	109%	37%	18%	12%
Gcd (No Bag Store)	109%	134%	146%	148%
Prime (Optimal)	115%	73%	46%	30%
Prime (No Bag Store)	115%	91%	75%	67%
Fibonacci (Optimal)	125%	85%	59%	39%
Fibonacci (No Bag Store)	125%	>200%	>200%	>200%

Figure 5.15: Experimental results with and without bag constraint store

instead forces goal executions to iterate through many more shared memory locations in the constraint store, thus increasing number of failed concurrent execution due to false overlaps. This explains why results (no ARV and no Indexing) worsens with more goal threads executing in parallel.

5.6.3 Disabling Bag Constraint Store

Figure 5.15 illustrates experiment results conducted without the use of **bag constraint stores and iterators** versus our optimal results from Section 5.6.1. Here, we highlight only the Gcd, Prime and Fibonacci examples as all others show little significant changes in scalability and performance. Without bag constraint store and iterators, Gcd and Fibonacci perform worse when more goal threads are executed in parallel. Even though Prime demonstrate better performance with more goal threads, its scalability with more threads is still less impressive than our optimal results.

These results are not entirely surprising, since Gcd, Fibonacci and Prime are indeed CHR problems where CHR goals likely share overlapping sets of potential candidates for partner constraints. For instance, consider the Fibonacci rule *fibo4*:

$$\mathit{fibo4} \ @ \ \mathit{Fibo}(x), \mathit{Fibo}(y) \Longleftrightarrow \mathit{Fibo}(x+y)$$

An active goal $Fibo(x1)\#n$ is free to match with any $Fibo$ constraint (variable y of rule head $Fibo(y)$ is unbounded), as such if all parallel goals iterate through potential candidate $Fibo$ constraints in the same order, they will frequently select overlapping constraints. Hence, more computation time is wasted for synchronizing between parallel goal threads (STM roll back and continue search for another available partner). As the experiment results in this section show, such unnecessary synchronization procedures are avoided in our optimal configuration by the use of bag constraint stores and iterators. The Gcd example CHR rule *gcd1* and the Prime example rule *prime3* also shares this similarly with the Fibonacci example rule *fibo4*.

	1 Thread	2 Threads	4 Threads	8 Threads
Merge Sort (Optimal)	121%	94%	70%	52%
Merge Sort (Unordered Goals)	121%	158%	190%	>200%
Gcd (Optimal)	109%	37%	18%	12%
Gcd (No Bag Store)	109%	62%	41%	28%

Figure 5.16: Experimental results, with domain specific goal ordering disabled

5.6.4 Disabling Domain Specific Goal Ordering

Figure 5.16 illustrates our experiment results without the domain specific goal ordering optimization. In our examples, only Mergesort and Gcd examples specifies a goal ordering[16], hence disabling goal ordering will only affect these two examples.

The results show that without goal ordering, Mergesort without goal ordering does not scale with increasing goal threads. This confirms that the optimal goal ordering reduces number of conflicting concurrent goal executions and number of *Next* comparisons (see [Lam, 2010] for details). Gcd without goal ordering still performs decently, but without the super linear speed up it experiences with goal ordering.

Bibliography

Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4, 1999.

Gregory J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, The University of Melbourne, 2005.

Thom Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence analysis. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 113–127. Springer-Verlag, 2005.

Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009. ISBN 0-521-87776-8.

GHC. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

Christian Holzbaur, Maria J. García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *TPLP*, 5(4-5):503–531, 2005.

Edmund S.L. Lam. *Parallel Execution of Constraint Handling Rules – Theory, Implementation And Application*. PhD thesis, National University of Singapore, Singapore, 2010.

Tom Schrijvers. Analyses, optimizations and extensions of Constraint Handling Rules: Ph.D. summary. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 435–436. Springer-Verlag, 2005.

Martin Sulzmann and Edmund S. L. Lam. Parallel execution of multi-set constraint rewrite rules. In *Proc. of PPDP'08*, pages 20–31. ACM Press, 2008.

[16]For Mergesort, *Next* goals are stacked, *Merge* goals are queued. For Gcd, goals are queued.

Martin Sulzmann, Edmund S. L. Lam, and Simon Marlow. Comparing the performance of concurrent linked-list implementations in haskell. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 37–46, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-417-1. doi: http://doi.acm.org/10.1145/1481839.1481845.

Part IV

Formal Analysis of CHR

Chapter 6

Computational Complexity

Author:	Jon Sneyers
Thesis Title:	Optimizing Compilation and Computational Complexity of Constraint Handling Rules
School:	K.U.Leuven, Belgium
Publication Year:	2008

"Computer Science is no more about computers than astronomy is about telescopes."

"The question of whether Machines Can Think ... is about as relevant as the question of whether Submarines Can Swim."

— Edsger Dijkstra (1930–2002)

Foreword

In this chapter we study the complexity of CHR. We start as follows. After a brief introduction to computational complexity theory in Section 6.1, we introduce a new model of computation, the CHR machine, and compare it with the standard models of computation — Turing machines and RAM machines. We define CHR machines in Section 6.2. Next, in Section 6.2.1 we show the unsurprising result that CHR machines are Turing-complete. Then, in Section 6.2.2 we define the time and space complexity of CHR machines.

A section on identifying syntactical subsets of CHR programs that are still Turing-complete ([Sneyers, 2008b, Ch.10.3]) has been omitted, because it has been superseded by more recent work [Sneyers, 2008a, Gabbrielli et al., 2010]. Also note that Listing 10.8 in [Sneyers, 2008b] is wrong (the last line will cause problems in case of a jump). This "bug" is no coincidence: this class of programs (single-headed CHR without functors) has meanwhile been shown to be non-Turing-complete by Gabbrielli et al. [2010].

We will derive a general complexity meta-theorem in Section 6.3 which we use to show a complexity-wise completeness result: we demonstrate that it is possible to implement *any* algorithm in CHR in an efficient way, i.e. with the best known time and space complexity. In Section 6.3.4 we investigate to what extent the result can be 'ported' to other declarative languages, and in Section 6.3.5 we examine the constant factors hidden behind the asymptotic big O notation.

6.1 Introduction to Complexity Theory

The theory of computation is one of the most fundamental branches of computer science. In a nutshell, it is the study of whether or not certain problems can be algorithmically solved, both in principle and in practice. The field of computability theory deals with the question of which problems can be solved algorithmically *in principle*, that is, if there are no time and space constraints. One of the most important results in computability theory is the undecidability of the *halting problem* [Turing, 1936]. In the field of (computational) complexity theory, the question is whether a problem can be algorithmically solved *in practice*. Although there may be an algorithm that solves the problem in principle, it could very well take such a huge amount of time or space — say, it takes longer than the age of the universe to complete the computation, or more bits than the number of particles in the universe — that it is essentially useless in practice. Complexity theory is mainly concerned with the scalability of algorithms, that is, the correlation between the size of the problem instance and the amount of computational resources needed by the algorithm. The most famous open problem in complexity theory is the question of whether the complexity classes **P** and **NP** coincide or not.

Both in computability theory and in computational complexity theory, idealized formal models of computation are constructed in order to define the notions of "algorithm" and "computational resource" in a precise way. In Section 6.1.1 we discuss three well-known models of computation: the Turing machine, the RAM (random access memory) machine, and the Minsky machine. (In Chapter 6.2 we will introduce a fourth model of computation, called the CHR machine.) Section 6.1.2 briefly discusses some elementary notions of computability theory; in particular, the concept of Turing-completeness and the Church-Turing thesis. Finally, in Section 6.1.3, we define the notions of time and space complexity in the different models of computation.

6.1.1 Models of computation

Different models of computation have been proposed for different purposes. The well-known Turing machine (Section 6.1.1) is conceptually very simple, which makes it very suitable for theoretical investigation. It also has a non-deterministic variant (Section 6.1.1). An even simpler model of computation is the Minsky machine (Section 6.1.1). The Random Access Memory machine (Section 6.1.1) is somewhat more complicated, but it more closely models realistic computers.

Turing machines

The Turing machine, originally introduced by Alan Turing [1936], is the prototypical computational model which is still widely used in computability and complexity theory. We use the single-tape definition of Hopcroft et al. [2001]:

Definition 6.1.1 (Turing machine)**.** *A (deterministic) Turing machine is a 6-tuple* $M = \langle Q, \Sigma, q_0, b, F, \delta \rangle$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of symbols, the tape alphabet;*
- $q_0 \in Q$ *is the initial state;*
- $b \in \Sigma$ *is the blank symbol;*

- $F \subseteq Q$ *is the set of accepting final states;*
- $\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{\texttt{left}, \texttt{right}\}$ *is a partial function, called the transition function; "*`left`*" and "*`right`*" represent the direction of the tape head move.*

In the literature definitions sometimes differ subtly, usually without an impact on the computational power or time complexity. For example the set $\{\texttt{left}, \texttt{right}\}$ is often extended to allow the machine to stay on the same tape cell.

A Turing machine M operates on an infinite tape of cells. Each cell contains a symbol of the tape alphabet Σ. The tape is assumed to be arbitrarily extendible to the left and the right. A head is positioned on a particular cell of the tape, can read and write a symbol in that cell and can move left and right.

Operation starts in the initial state q_0 on a tape which contains a finite string of symbols (called the *input*), and the head is positioned on the left-most input symbol. Execution proceeds by considering the current state q and the symbol s that is under the head. Then:

- Either (q, s) is a member of the domain of δ and $\delta(q, s) = (q', s', X)$. The effect then is that the current state of M changes to q', the head overwrites the value s in the cell under it with s' and next the head either moves to the left or the right depending on whether $X = \texttt{left}$ or $X = \texttt{right}$.
- Or (q, s) is not part of the domain of δ. Execution stops. If $q \in F$, the input is *accepted*, otherwise it is *rejected*.

We represent an execution state of a Turing machine $M = \langle Q, \Sigma, q_0, b, F, \delta \rangle$ as a 4-tuple $\sigma^{\mathrm{TM}} = \langle q, P, s, N \rangle$, representing the *current state* $q \in Q$, the *current symbol* $s \in \Sigma$, the *previous symbols* P and the *next symbols* N which are sequences of symbols. We use $\sigma_0^{\mathrm{TM}}, \sigma_1^{\mathrm{TM}}, \ldots$ to denote Turing machine execution states, and $\Sigma^{\mathrm{TM}}(M)$ to denote the set of all execution states of M. Given an input $I = [f|N]$ (a sequence of symbols), the initial execution state is $\sigma_0^{\mathrm{TM}} = \langle q_0, [], f, N \rangle$. The transition rules are:

$$
\begin{array}{lrcll}
(1) & \langle q, [p|P], s, N \rangle & \rightarrow & \langle q', P, p, [s'|N] \rangle & \text{if} \quad \delta(q, s) = (q', s', \texttt{left}) \\
(2) & \langle q, P, s, [n|N] \rangle & \rightarrow & \langle q', [s'|P], n, N \rangle & \text{if} \quad \delta(q, s) = (q', s', \texttt{right}) \\
(3) & \langle q, [], s, N \rangle & \rightarrow & \langle q', [], b, [s'|N] \rangle & \text{if} \quad \delta(q, s) = (q', s', \texttt{left}) \\
(4) & \langle q, P, s, [] \rangle & \rightarrow & \langle q', [s'|P], b, [] \rangle & \text{if} \quad \delta(q, s) = (q', s', \texttt{right})
\end{array}
$$

The machine operates by exhaustively applying the transition rules on the initial execution state. Note that the size of execution states is unbounded but finite. In execution state $\sigma_0^{\mathrm{TM}} = \langle q, P, s, N \rangle$, the contents of the tape is the sequence $\mathit{tape}(\sigma_0^{\mathrm{TM}}) = \mathit{reverse}(P) \mathbin{+\!\!+} [s] \mathbin{+\!\!+} N$, where *reverse* reverses a sequence (i.e. $\mathit{reverse}([a_1, a_2, \ldots, a_{n-1}, a_n])$ equals $[a_n, a_{n-1}, \ldots, a_2, a_1]$). If the final execution state is of the form $\langle q_f, P, s, N \rangle$, the Turing machine has *accepted* the input I if $q_f \in F$. If $q_f \in Q \setminus F$, the input is *rejected*. When the input is accepted, the *output* of the Turing machine is the tape contents in the final execution state; when the input is rejected, the output is undefined. Given a Turing machine M and input I, we denote the corresponding derivation by $\mathit{deriv}_M(I)$. If the machine terminates on input I, we denote the output by $M(I) = \mathit{tape}(\sigma_f^{\mathrm{TM}})$, where σ_f^{TM} is the last state in $\mathit{deriv}_M(I)$.

Non-deterministic Turing machines

In deterministic Turing machines, there is exactly one next step (or none, for the configurations in which the machine terminates) for every combination of a state and an input

symbol. Non-deterministic Turing machines do not have this restriction: there can be different actions for the same state and input symbol.

Definition 6.1.2 (non-deterministic TM)**.** *A non-deterministic Turing machine is a 6-tuple $M = \langle Q, \Sigma, q_0, b, F, \delta \rangle$ which is defined as in Definition 6.1.1, except for δ, which is an arbitrary relation (not necessarily a function):*

$$\delta \subseteq (Q \times \Sigma) \times (Q \times \Sigma \times \{\texttt{left}, \texttt{right}\})$$

A non-deterministic Turing machine works just like a deterministic one, except that whenever there are $n > 1$ possible actions to be taken, n duplicates of the machine are made, and every choice is executed in parallel. As soon as one of the execution branches halts in an accepting final states, the entire non-deterministic Turing machine stops and accepts the input (the output is the tape content of the accepting branch). When all execution branches have halted in a rejected final state, the entire non-deterministic Turing machine stops and rejects the input. Otherwise, the machine does not terminate.

RAM machines

The Random Access Memory (RAM) machine [Aho et al., 1975] closely models the basic features of traditional sequential computers. In the literature many variations of the RAM machine have been considered. We investigate two different RAM machines. The first is a RAM machine with simple Peano arithmetic operations and the second has the standard arithmetic operations as they are implemented on today's computers.

Common architecture. A RAM machine consists of three components: the central processing unit (CPU), the program, and the random-access memory (RAM). The memory consists of an infinite number of cells, or registers, which are labeled with a natural number which is called its *address*. If a register is *initialized*, it contains a *value*, which is an integer number. We use $A, A_1, A_2, \ldots$ to represent the memory addresses and $[\![A]\!]$ to denote the value of the register at address A.

The program consists of a sequence of instructions. The program instructions are labeled with successive natural numbers. We use $L, L_1, \ldots$ to denote program instruction labels. The CPU follows a fetch-and-execute cycle. It has a program counter PC that is initialized to the first program instruction label. This program counter contains the label of the next program instruction to be executed. The CPU fetches the instruction and performs the corresponding operations.[1] This involves setting the program counter to the next instruction, by default the successor of the current address. Table 6.1 lists the instructions supported by the standard RAM machine. The Peano-arithmetic RAM machine uses a subset of these instructions.

Definition 6.1.3 (Peano-arithmetic RAM machine)**.** *A Peano-arithmetic RAM machine consists of a program and a working memory as described above. The program instructions are* `inc`*,* `dec`*,* `clr`*,* `jmp`*,* `cjmp`*, and* `halt` *(see Table 6.1).*

This corresponds to the definition used by Savage [1998]. Indirect addressing is not supported, so all registers that are used in a program are supposed to be initialized in advance. All copying, addition and subtraction has to be done by repeated use of the `inc` and `dec` instructions. This makes the Peano-arithmetic RAM less practical, as actual computers do provide instructions for addition and subtraction.

[1] If an illegal instruction is encountered, the machine halts. Examples of illegal instructions are division by zero, jump to a non-existent label, instructions referring to registers with a negative address, etc.

Instruction			*Effect*
`inc`	A		$[\![A]\!] \leftarrow [\![A]\!] + 1$
`dec`	A		$[\![A]\!] \leftarrow [\![A]\!] - 1$
`clr`	A		$[\![A]\!] \leftarrow 0$
`jmp`	L		$PC \leftarrow L$
`cjmp`	A	L	$PC \leftarrow L$ if $[\![A]\!] = 0$
`halt`			Halt execution of the RAM machine
`init`	A		Initialize the register at address $[\![A]\!]$ to zero
`cnst`	B	A	$[\![A]\!] \leftarrow B$
`add`	A_2	A_1	$[\![A_1]\!] \leftarrow [\![A_1]\!] + [\![A_2]\!]$
`sub`	A_2	A_1	$[\![A_1]\!] \leftarrow [\![A_1]\!] - [\![A_2]\!]$
`mul`	A_2	A_1	$[\![A_1]\!] \leftarrow [\![A_1]\!] * [\![A_2]\!]$
`div`	A_2	A_1	$[\![A_1]\!] \leftarrow [\![A_1]\!]/[\![A_2]\!]$ if $[\![A_2]\!] \neq 0$
`mov`	A_2	A_1	$[\![A_1]\!] \leftarrow [\![A_2]\!]$
`imv`	A_2	A_1	$[\![A_1]\!] \leftarrow [\![[\![A_2]\!]]\!]$
`mvi`	A_2	A_1	$[\![[\![A_1]\!]]\!] \leftarrow [\![A_2]\!]$

Table 6.1: Instruction set of the RAM machine

Definition 6.1.4 (standard RAM machine)**.** *A standard-arithmetic RAM machine consists of a program and a working memory as described above. The program instruction set is given in Table 6.1.*

This instruction set is similar to the definition of Aho et al. [1975], and resembles more closely actual computers. The `inc`, `dec`, and `clr` instructions are redundant: they can be implemented using `add`, `sub`, and `cnst`. Without loss of generality, we assume the instructions that refer to two (`add`, `sub`, `mul`, `div`, `mov`) or three (`imv`, `mvi`) memory cells refer to two or three *different* cells. For example, the instruction "`add` a a" should be rewritten to "`mov` a t ; `add` t a" where t is some temporary register.

By convention, particular memory cells are initialized in advance and contain the input, while other (or the same) memory cells are meant to contain the output. Without loss of generality we assume that the program initializes all additional registers (using the `init` instruction) before they are used.

Minsky machines

An even simpler version of the Peano-arithmetic RAM machine was proposed by Minsky [1967]. Minsky machines have only two registers, that can hold natural numbers. There are only two instructions: `succ(`r_i`)`, which increases the value of r_i by one, and `decjump(`r_i`,`p_j`)`, which decreases the value of r_i by one if it is nonzero, and otherwise jumps to program line p_j. A program consists of a numbered sequence of instructions and unless there is a jump, the program continues with the next instruction in the sequence. Some program lines may be empty; in that case the machine halts (in some formulations a third instruction `halt` is used to represent these empty lines). Without loss of generality we can assume that both registers are initialized to zero.

6.1.2 Computability

Computability theory studies the boundary between the problems that can be solved algorithmically, and those that cannot. An algorithm that solves a problem (also called an

effective method or *effective procedure*) is a method consisting of steps that may be described as mechanical operations; a method which, if followed rigorously, has the following properties:

- the method always terminates after a finite number of steps;
- the method always gives the correct answer and never a wrong answer;
- the method works for all instances of the problem.

Note that this is not a mathematically precise definition — the notion of effective procedure is somewhat vague and intuitive, in particular because it is not specified what kind of operations "may be described as mechanical operations". The models of computation of the previous section were formulated in an attempt to formalize this notion of an algorithm.

Church-Turing thesis

The Church-Turing thesis can be formulated as follows: "A problem can be algorithmically solved if and only if there is a Turing machine that solves it." In other words, the notion of computability by a Turing machine and the intuitive notion of effective computability are in fact the same. This is not a mathematical statement that can be proven correct formally because it is a statement about the informal notion of an algorithm.

It is supported by the equivalence of Turing machines and λ-calculus, a formal system introduced by Alonzo Church and Stephen Kleene in the 1930s. Both formalisms were introduced independently to capture what is computable. They are very different formalisms, originating from a different approach to computability: the "mechanical" approach in the case of Turing machines and the "mathematical" or "functional" approach in the case of λ-calculus. Still, both formalisms were shown to be equivalent — which indicates that either formalism really corresponds to the intuitive notion of "effective computability".

According to the Church-Turing thesis, all 'reasonable' models of computation are equivalent to the Turing machine in the sense that they can all solve the same problems. Today, about 70 years after its first formulation, the Church-Turing thesis is widely believed to hold. Indeed, for all reasonably strong models of computation that have been proposed in the literature, it has been proved that they are (computationally) equivalent to the Turing machine. Clearly, models of computation can also be defined that are weaker than Turing machines — for example, finite automata.

Turing-completeness

A model of computation is called *Turing-complete* if it has (at least) the same computational power as Turing machines, that is, every Turing Machine can be simulated in the model. A model of computation is called *Turing-equivalent* if it has precisely the same computational power as Turing machines, i.e. it is Turing-complete and every program of the model can also be simulated on a Turing machine. All 'reasonable' Turing-complete models of computation are also Turing-equivalent[2], a fact that adds support to the Church-Turing thesis.

[2] Some contrived models of computation that are not intended to model some physically realizable computational device are not 'reasonable'. For example, so-called *oracle machines* are more powerful than plain Turing machines if the oracle is sufficiently powerful, for instance capable of solving the halting problem in a single step. Another example is the so-called *Zeno machine*, in which each computational step takes half the time of the previous step — such a machine can perform a countably infinite number of steps in finite time.

A *universal Turing machine* is a Turing machine program that can "run" any arbitrary Turing machine: if the tape is initialized with some encoding of a Turing machine program followed by an input tape for that program, the final output on the tape is exactly the same as the output the encoded program would give. The existence of such universal machines is arguably one of the most important results in the whole of computer science, and it is due to Turing [1936]. The von Neumann architecture and concepts like program-as-data, subroutines, compilers, operating systems, ... were directly influenced by the existence of universal machines.

6.1.3 Computational complexity theory

Complexity theory investigates the amount of computational resources needed to execute an algorithm. The two most important computational resources are time (execution time — number of steps) and space (size of the required memory — number of tape cells or registers).

One of the seminal papers that founded and shaped the field of computational complexity theory was [Hartmanis and Stearns, 1965]. Relatively recently, in 1988, a letter surfaced[3] that indicates that the core idea and importance of complexity theory was already recognized by Kurt Gödel as early as 1956.

Time complexity

The worst-case time complexity of an algorithm or program is defined as a function of the size of the input. For a given input size, the worst-case time complexity is the maximal number of execution steps needed for executing the program on arbitrary input of that size.

It is important to make explicit, for a given problem, how the size of a problem instantiation (the input size) is to be measured. For example, if the problem is testing whether or not a number is a prime, one could count the number itself as the size of the input or, alternatively, the number of bits (or decimal digits) needed to represent the number. Depending on which input size measure is used, the time complexity of the AKS primality test [Agrawal et al., 2004] is polylogarithmic or polynomial, respectively.

In some cases the related notions of average-case and best-case complexity may also be useful. Usually, we are interested in performance guarantees, for which the worst-case is most important. From now on, when we say "complexity", we implicitly mean the worst-case complexity.

For deterministic Turing machines, the length $\#deriv_M$ of a derivation $deriv_M$ is simply the number of steps in that derivation. Recall that for non-deterministic Turing machines, we consider the shortest possible derivation that leads to an accepting final state; if there are no accepting paths, the derivation length is defined to be zero.

Definition 6.1.5 (time complexity of Turing machines)**.** *The time complexity of a Turing machine $M = \langle Q, \Sigma, q_0, b, F, \delta \rangle$ is the maximal derivation length for inputs of a given size:*

$$\textsc{tmtime}_M(n) = \max\{\#deriv_M(I) \mid I \in \Sigma^* \text{ and } |I| = n\}$$

Definition 6.1.6 (time complexity of RAM machines)**.** *The time complexity* $\textsc{ramtime}_P(n)$ *of a RAM machine with program P is the maximal time needed for an execution starting*

[3] The letter can be found at http://www.contrib.andrew.cmu.edu/~hardt/godel.html.

with some input of size n. The time needed for a program execution is the sum of the times needed for every instruction that is executed. All instructions take constant time except for the arithmetic instructions `add`*,* `sub`*,* `mul`*, and* `div`*, which take $O(\log|x|)$ time where x is the largest (in absolute value) of the numbers involved in the arithmetic operation.*

Space complexity

The space complexity of an algorithm is the number of tape cells (or bits) it needs, in the worst case, during its execution. As time complexity, it is defined as a function of the input size.

Definition 6.1.7 (space complexity of Turing machines)**.** *The space complexity of a Turing machine M is the maximal tape size used in derivations for inputs of a given size:*

$$\text{TMSPACE}_M(n) = \max\{\#tape(\sigma) \mid \sigma \in deriv_M(I) \text{ and } I \in \Sigma^* \text{ and } |I| = n\}$$

Definition 6.1.8 (space complexity of RAM machines)**.** *The space complexity* $\text{RAMSPACE}_P(n)$ *of a RAM machine with program P is the maximum, over every execution starting with an input of size n, of the sum over all registers in the range $0, \ldots,$ maxaddr of the number of bits needed to represent the maximal value that it held. Unused registers within that range are charged one bit for the implicit value 0 stored in them.*

In practice, we often assume the registers to use at most a fixed number of bits. In that case, every instruction takes constant time and the space complexity is just *maxaddr*, the maximum register address reached in a computation.

In the following, we will assume RAM programs to use at least as much time as space. This is a nontrivial assumption given our definition of space complexity, but unless a program uses the registers in an unrealistically sparse way, it is not a very restrictive assumption. After all, every program can be rewritten to use memory in a dense way with only logarithmic time overhead, e.g. by explicitly storing address-value pairs using a balanced search tree.

Asymptotic complexity and big O notation

We are usually not interested in the exact time or space complexity of an algorithm, but only in its scalability. The practical reason is that real computers tend to become faster and faster and the amount of physical memory also tends to increase. Hence, if I have an algorithm which is a constant factor k slower than your algorithm, we can still handle the same range of input sizes — I just have to buy a more expensive computer that is k times as fast as yours, or wait until such computers become available. However, if my algorithm takes 2^n steps for input of size n and your algorithm takes n^2 steps, then there will always be inputs that you can handle but I cannot (in, say, one human lifetime), no matter how much faster my computer is compared to yours. A more theoretical reason to ignore constant factors is the so-called *linear speedup* theorem [Hartmanis and Stearns, 1965]: if a problem can be solved by a Turing machine in time $f(n)$, there is also a Turing machine that solves it in time $cf(n) + n + 2$, for any $c > 0$.

The notion of *asymptotic* complexity is used to compare the scalability of algorithms. A function $f(n)$ is asymptotically bounded by another function $g(n)$, which we denote by saying $f(n)$ is $O(g(n))$, if and only if

$$\exists N, c > 0 : \forall n > N : f(n) \leq cg(n)$$

Complexity	*Name*	*Example problem (algorithm)*
$O(1)$	constant	Accessing an arbitrary element of an array
$O(\log n)$	logarithmic	Searching in a sorted list of length n
$O(n)$	linear	Searching in an unsorted list of length n
$O(n\alpha(n))$	inverse Ackermann	Performing n union-find operations
$O(n \log n)$	quasilinear	Sorting a list of n elements (heapsort)
$O(n^2)$	quadratic	Sorting a list of n elements (insertion sort)
$O(n^3)$	cubic	Finding a shortest path in a weighted graph (where edge weights may be negative) with n nodes (Bellman-Ford algorithm)
$O(n^k)$	polynomial	Testing whether a number of n digits is prime (Agrawal-Kayal-Saxena primality test)
$O(k^n)$	exponential	Traveling salesman problem (dynamic programming)
$O(n!)$	factorial	Traveling salesman problem (brute force)

Table 6.2: Common asymptotic complexities

If algorithm A has complexity $f(n)$ and algorithm B has complexity $g(n)$, and $f(n)$ is $O(g(n))$, then we say algorithm A is at least as efficient as algorithm B. If both $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$, we say that $f(n)$ is $\Theta(g(n))$.

Table 6.2 lists some common asymptotic time complexities, ordered from most efficient to least efficient. In this table, n denotes the input size and k denotes some constant. We also use the notation $\tilde{O}(f(n))$ as a shorthand for $O(f(n) \log^k f(n))$, for some fixed k. In this notation, polylogarithmic factors are ignored. Polylogarithmic factors are often not that important since any function that is $\tilde{O}(n^k)$ is also $O(n^{k+\epsilon})$, for arbitrary small $\epsilon > 0$.

Amortized complexity analysis

The execution of many algorithms, in particular data structure algorithms, consists of a sequence of operations. In most circumstances (the exception being real-time programs) we are only interested in the total time, not in the time per operation.

Sometimes the time per operation varies a lot, and as a result, the worst-case time per operation would seem rather bad. However, if the worst case only happens demonstrably rarely, the worst-case time for sequences of operations can be much better than the naive bound obtained by multiplying the length of the sequence by the worst-case time per operation.

The *amortized complexity* of an operation is the average time needed for the operation, over a worst-case *sequence* of operations. As a simple example, consider a data structure which has two operations: `push(`x`)`, which puts some item x at the top of a stack, and `popall`, which removes all items from the stack while printing them. Clearly, if there are n elements on the stack, the time needed for the `popall` operation is $O(n)$. Given a sequence of k operations, the stack size may be as large as k, so the total time is $O(k^2)$. Using amortized complexity analysis, we can improve this bound. Both operations take only constant amortized time.

This can be seen as follows. We assign a *potential*, a positive number, to every possible state of the stack data structure. The amortized time for an operation is its actual time plus the net increase in potential caused by it. The actual time for a sequence of operations is then the total amortized time minus the total net increase in potential. We define the

potential of a stack simply as the number of elements it contains. At the beginning of a sequence of operations we have an empty stack, so the initial potential is zero. Since the potential cannot be negative, the amortized time complexity is an upper bound for the actual time complexity. The `push` operation indeed has a constant amortized time complexity: adding one element to the stack takes constant time, and the potential of the stack is increased by one, which is also a constant. The `popall` operation takes n time units if the stack has size n, but the increase in potential it causes is $-n$, so its amortized complexity is also constant. We can conclude that any sequence of `push` and `popall` instructions of length k takes only $O(k)$ time.

In [Sneyers, 2008b], chapter 5.2.3, we give a more interesting example of amortized complexity analysis when discussing the time complexity of the Fibonacci heap algorithm of Fredman and Tarjan [1987].

Relation between RAM machines and Turing machines

It is easy to see that RAM machines are Turing-complete. Both kinds of RAM machines can simulate a T-time Turing machine in $O(T)$ time. Also, both kinds of RAM machines are Turing-equivalent. The main difference between the two RAM machines is the time complexity that can be achieved when simulating them on a Turing machine. According to Savage [1998], a T-time Peano-arithmetic RAM using S registers can be simulated on a Turing machine in $O(ST \log^2 S)$ time. The standard RAM is also polynomially related to the Turing machine, although it is more expensive to simulate on a TM. According to Aho et al. [1975], a standard RAM machine with time complexity T can be simulated on a multi-tape TM in $O\big((T \log T \log\log T)^2\big)$ time. Simulating a multi-tape TM on a single-tape TM squares the time complexity [Hartmanis and Stearns, 1965], so we have:

Lemma 6.1.9. *Any standard RAM machine with time complexity T can be simulated on a Turing machine with time complexity $\tilde{O}(T^4)$.*

6.2 CHR Machines

We define a CHR machine as follows:

Definition 6.2.1 (CHR machine)**.** *A CHR machine is a tuple $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$. The host language $\mathcal{H}$ defines a built-in constraint theory $\mathcal{D}_\mathcal{H}$, $\mathcal{P}$ is a CHR program, and $\mathcal{VG} \subseteq \mathcal{G}_\mathcal{P}^\mathcal{H}$ is a set of valid goals, such that $\mathcal{P}$ is a $\Delta_{\omega_t}^\mathcal{H}$-deterministic CHR program for input $\mathcal{VG}$. The machine takes an input query $\mathbb{G} \in \mathcal{VG}$ and executes a derivation $d \in \Delta_{\omega_t}^\mathcal{H}|_\mathbb{G}$.*

To fully specify a CHR machine, we need to give the host language, the CHR program it uses, and the set of queries that are allowed as input. Note that the above definition heavily restricts the class of allowed CHR programs: they have to be $\Delta_{\omega_t}^\mathcal{H}$-deterministic for valid input queries.

In [Sneyers, 2008b], chapter 12, we consider more general types of CHR machines. For now, the above definition suffices and it is considerably simpler. In the terminology of [Sneyers, 2008b], chapter 12, CHR machines as defined above are both *deterministic* and *abstract* (in the sense that they follow the abstract operational semantics ω_t).

Terminology. Since we only allow deterministic CHR programs, we will assume that there is exactly one derivation corresponding to every goal. If there are different derivations, we arbitrarily pick one — the resulting final state has to be unique anyway, and in

Listing 6.1: TMSIM: Turing machine simulator

```
r1 @ delta(Q,S,Q2,T,left), adj(L,C)
  \ state(Q), cell(C,S), head(C)
  <=> L \== null | state(Q2), cell(C,T), head(L).
r2 @ delta(Q,S,Q2,T,right), adj(C,R)
  \ state(Q), cell(C,S), head(C)
  <=> R \== null | state(Q2), cell(C,T), head(R).
r3 @ delta(Q,S,Q2,T,left)
  \ adj(null,C), state(Q), cell(C,S), head(C)
  <=> cell(L,b), adj(null,L), adj(L,C),
      state(Q2), cell(C,T), head(L).
r4 @ delta(Q,S,Q2,T,right)
  \ adj(C,null), state(Q), cell(C,S), head(C)
  <=> cell(R,b), adj(C,R), adj(R,null),
      state(Q2), cell(C,T), head(R).
fail @ nodelta(Q,S), reject(Q), state(Q), cell(C,S), head(C)
  <=> fail.
```

case it is successful, there cannot even be different derivation lengths. Therefore we will often talk about *the* derivation for a given goal as a shorthand for "an arbitrary derivation". We use $deriv_{\mathcal{M}}(\mathbb{G})$ to denote the derivation for goal $\mathbb{G}$. If the derivation $d = deriv_{\mathcal{M}}(\mathbb{G})$ is finite, we say the machine *terminates* for goal $\mathbb{G}$, with *output state* $\mathcal{M}(\mathbb{G}) = \langle \mathbb{G}', \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ which is the last state of d. The machine *accepts* the input $\mathbb{G}$ if d is a successful derivation and *rejects* $\mathbb{G}$ if d is a failed derivation. If d is an infinite derivation, we say the machine *does not terminate*.

A *CHR(X) machine* is a CHR machine with host language X, i.e. of the form $(X, \_, \_)$. We use Φ to denote *no host language*: the built-in constraint theory $\mathcal{D}_\Phi$ defines only the basic constraints `true` and `fail`, and syntactic equality and inequality (only to be used as an *ask*-constraint). This implies that the **Solve** transition can only be used once (to add `fail`). The only data types are constants, and variables that cannot be bound. A *CHR-only machine* is a CHR(Φ) machine.

A *sufficiently strong* host language $\mathcal{H}$ is a host language whose built-in constraint theory $\mathcal{D}_\mathcal{H}$ defines at least `true`, `fail`, `==` and `\==`, the integer numbers and the arithmetic operations for addition, subtraction, multiplication and integer division. Clearly, most realistic host languages are sufficiently strong. Prolog, for instance, defines `true`, `fail`, `==` and `\==`, and allows integer arithmetic using `is/2`.

6.2.1 CHR machines are Turing-complete

We now show that CHR-only machines are Turing-complete (cf. Section 6.1.2), i.e. they have at least the computational power of Turing machines. In order to prove this, it suffices to construct a CHR machine which corresponds to a universal Turing machine, i.e. a CHR machine which can simulate any Turing machine.

Consider the CHR program TMSIM shown in Listing 6.1 and the corresponding CHR-only machine $\mathcal{M}_{\mathrm{TM}} = (\Phi, \mathsf{TMSIM}, \mathcal{VTMG})$. We postpone the definition of $\mathcal{VTMG}$ for now. The program TMSIM simulates Turing machines (see Section 6.1.1).

Description. The TMSIM program defines the following constraints:

`delta`/5 encodes the transition function δ (the Turing machine program) in the obvious way: `delta(`q`,`s`,`q'`,`s'`,`d`)` means that $\delta(q, s) = (q', s', d)$;

`nodelta`/2 encodes the domain on which δ is undefined;

`reject`/1 encodes the set of non-accepting final states $Q \setminus F$;

`state`/1 contains the current state;

`head`/1 contains the identifier of the cell under the head;

`cell`/2 represents a tape cell. The first argument is the unique identifier of the cell. The second argument is the symbol in the cell.

`adj`/2 encodes the order of the tape cells. The constraint `adj`(A, B) should be read: "the right neighbor of the tape cell with identifier A is the tape cell with identifier B".

The special cell identifier `null` is used to refer to a not yet instantiated cell. The rules r_3 and r_4 take care of extending the tape as needed. The first four rules of TMSIM correspond to the four Turing machine transition rules.

A simulation of the execution of a Turing machine M proceeds as follows. The tape input is encoded as `cell`/2 constraints and `adj`/2 constraints. The identifier of the cell to the left of the left-most input symbol is set to `null` and similarly for the cell to the right of the right-most input symbol. The transition function δ of M is encoded in multiple `delta`/5 constraints. All these constraints are combined in the initial query together with the constraint `state`(q_0) where q_0 is the initial state of M and the constraint `head`(c_1) where c_1 is the identifier of the cell representing the left-most input symbol. Every rule application of the first four rules of TMSIM corresponds directly to a Turing machine transition.

If no more (Turing machine) transitions can be made, the last rule is applicable if the current state is non-accepting. In that case, the built-in constraint `fail` is added, which leads to a failure state. If the Turing machine ends in an accepting final state, the CHR program ends in a successful final state.

Correctness proof. We define a function tm_to_chr which produces a query for TMSIM, given a Turing machine and an input tape. It is defined as follows: given a Turing machine $M = \langle Q, \Sigma, q_0, b, F, \delta \rangle$ and an input tape $I = [i_1, \ldots, i_n]$,

$$\mathsf{tm\_to\_chr}(M, I) = \mathsf{prog\_to\_chr}(M) \cup \mathsf{tape\_to\_chr}(I) \cup \{\texttt{state}(q_0), \texttt{head}(c_1)\}$$

where $\mathsf{prog\_to\_chr}(M)$ is defined as follows:

$$\begin{aligned}\mathsf{prog\_to\_chr}(M) = \; & \{\texttt{delta}(q,\, s,\, q',\, s',\, d) \mid (q, s) \in Q \times \Sigma \text{ and } \delta(q, s) = (q', s', d)\} \\ & \cup \{\texttt{nodelta}(q,\, s) \mid (q, s) \in Q \times \Sigma \text{ and } \delta(q, s) \text{ is undefined}\} \\ & \cup \{\texttt{reject}(q) \mid q \in Q \setminus F\}\end{aligned}$$

and $\mathsf{tape\_to\_chr}(I)$ is defined as follows:

$$\mathsf{tape\_to\_chr}(I) = \{\texttt{adj}(\texttt{null},c_1)\} \cup \bigcup_{j=1}^{n} \{\texttt{cell}(c_j,i_j), \texttt{adj}(c_j,c_{j+1})\}$$

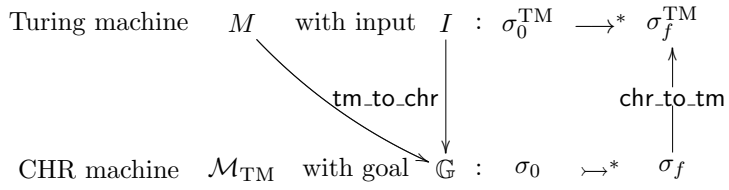


Figure 6.1: Simulation of Turing machines on a CHR machine

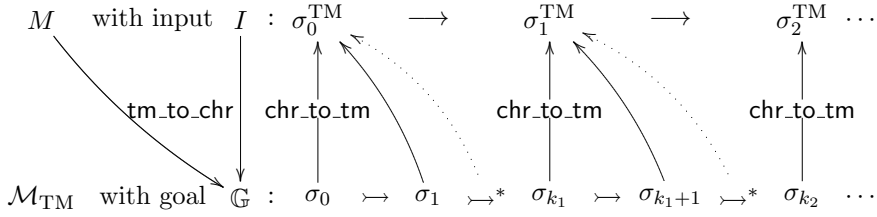


Figure 6.2: States of $\mathcal{M}_{\mathrm{TM}}$ and corresponding Turing machine states

where $c_{n+1} = \texttt{null}$ and the other c_j are unique cell identifiers. Clearly, tm_to_chr can be computed in time linear in the size of the input tape plus the size of the domain of the Turing machine program. We now define $\mathcal{VTMG}$ as follows:

$$\mathcal{VTMG} = \{\mathsf{tm\_to\_chr}(M, I) \mid M \text{ is a Turing machine for which } I \text{ is an input tape}\}.$$

Lemma 6.2.2. *$\mathcal{M}_{\mathrm{TM}}$ is indeed a CHR machine, that is, the CHR program* TMSIM *is $\Delta^{\Phi}_{\omega_t}$-deterministic for input queries from $\mathcal{VTMG}$.*

Proof. Clearly, the rules of TMSIM maintain a valid tape representation and the invariant that there is at most one `head`/1 constraint and one `state`/1 constraint. For deterministic Turing machines the first two arguments of `delta`/5 functionally determine the other three arguments. Hence, since valid input corresponds to a deterministic Turing machine, the rules of TMSIM are mutually exclusive. As a result, the **Apply** transitions of a derivation are determined, and only the order of the **Introduce** transitions may vary. However, it can easily be verified that the order of **Introduce** transitions cannot affect the derivation result and it can only affect the derivation length for failing derivations. □

We define a function chr_to_tm which returns a Turing machine execution state, given an execution state for a CHR machine: $\mathsf{chr\_to\_tm} : \Sigma^{\mathrm{CHR}} \to \Sigma^{\mathrm{TM}}$:
$\mathsf{chr\_to\_tm}(\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle q, P, s, N \rangle$ where q, P, S, and N are such that $\mathbb{G} \uplus \mathbb{S} = \mathsf{prog\_to\_chr}(M) \cup \{\mathtt{state}(q), \mathtt{head}(c), \mathtt{cell}(c,s)\} \cup \mathsf{tape\_to\_chr}(\mathit{tape}(\langle q, P, s, N \rangle))$.
It can easily be verified that this is indeed a function, and that it can be computed in time linear in the tape size.

We now show that $\mathcal{M}_{\mathrm{TM}}$ can simulate every Turing machine (cf. Figure 6.1).

Theorem 8. *For any Turing machine M and input tape I :*

1. *M terminates on input I $\Leftrightarrow$ $\mathcal{M}_{\mathrm{TM}}$ terminates on input* $\mathsf{tm\_to\_chr}(M, I)$
2. *M accepts I $\Leftrightarrow$ $\mathcal{M}_{\mathrm{TM}}$ accepts* $\mathsf{tm\_to\_chr}(M, I)$
3. *M outputs* $\mathsf{chr\_to\_tm}(X)$ *on input I*
   $\Longleftrightarrow$ $\mathcal{M}_{\mathrm{TM}}$ outputs X on input $\mathsf{tm\_to\_chr}(M, I)$

Proof. Let $\sigma_0 = \mathit{initstate}(\mathsf{tm\_to\_chr}(M, I))$ be the initial state of $\mathcal{M}_{\text{TM}}$ with the input $\mathsf{tm\_to\_chr}(M, I)$. Note that $\mathsf{chr\_to\_tm}(\sigma_0) = \langle q_0, [], i_1, [i_2, \ldots, i_n] \rangle$ is the initial execution state of M. We observe the following:

Observation 1. For every $\mathcal{M}_{\text{TM}}$ transition $\sigma_i \rightarrowtail \sigma_{i+1}$, either

(a) $\mathsf{chr\_to\_tm}(\sigma_i) \rightarrow \mathsf{chr\_to\_tm}(\sigma_{i+1})$ is a TM transition for M, or

(b) $\mathsf{chr\_to\_tm}(\sigma_i) = \mathsf{chr\_to\_tm}(\sigma_{i+1})$.

This can be shown by induction on the number of CHR machine steps and case analysis on the transition rules of ω_t. The first case (a) holds if $\sigma_i \rightarrowtail \sigma_{i+1}$ by the **Apply** transition rule, where the applied rule is $\mathtt{r}x$. The second case (b) holds if $\sigma_i \rightarrowtail \sigma_{i+1}$ by the **Introduce** transition rule. Case (b) also holds when $\sigma_i \rightarrowtail \sigma_{i+1}$ by the **Apply** transition rule for the rule *fail*, which can only be followed by (a series of **Introduce** transitions followed by) a **Solve** transition which results in a failure state. The relation between the Turing machine derivation steps and the CHR machine steps is shown schematically in Figure 6.2.

Observation 2. If σ_i is a final state (for $\mathcal{M}_{\text{TM}}$), then $\mathsf{chr\_to\_tm}(\sigma_i)$ is a final state (for M). Furthermore, if σ_i is a successful final state, then $\mathsf{chr\_to\_tm}(\sigma_i)$ is an accepting final state, and if σ_i is a failure state, then $\mathsf{chr\_to\_tm}(\sigma_i)$ is non-accepting.

Observation 3. The sub-derivations corresponding to one Turing machine step are finite. In other words, all chains of the form

$$\mathsf{chr\_to\_tm}(\sigma_n) = \mathsf{chr\_to\_tm}(\sigma_{n+1}) = \mathsf{chr\_to\_tm}(\sigma_{n+2}) = \ldots$$

have a finite length. This follows from observation 1 and the definition of the **Introduce** transition, which decrements the size of the (finite) multiset representing the goal in CHR execution states.

Observation 4. If M terminates on input I, then the $\mathcal{M}_{\text{TM}}$ derivation on input $\mathsf{tm\_to\_chr}(M, I)$ is also finite. This follows from observation 1 and 3.

The three properties follow straightforwardly from the above observations. □

6.2.2 Complexity of CHR machines

Thus far we have only considered the computability of CHR machines. In the following chapters we will investigate the computational complexity of CHR machines. But first, we need to complete the definition of CHR machines of Section 6.2, and define what exactly we mean with the time and space complexity of a CHR machine.

Time complexity

We will use a rather simple and natural measure for the time complexity of CHR machines:

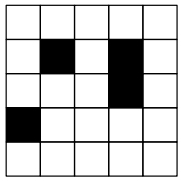
Definition 6.2.3 (time function)**.** *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$ and a valid goal $\mathbb{G}$, the time function $\mathit{chrtime}_{\mathcal{M}}$ returns the derivation length:*

$$\mathit{chrtime}_{\mathcal{M}} : \mathcal{VG} \rightarrow \mathbb{N} : \mathbb{G} \mapsto \#\mathit{deriv}_{\mathcal{M}}(\mathbb{G}).$$

Definition 6.2.4 (time complexity of CHR machines)**.** *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$ and assuming that host language constraints of $\mathcal{H}$ take constant time, the (worst-case) time complexity function* $\text{CHRTIME}_{\mathcal{M}}$ *is defined as follows:*

$$\text{CHRTIME}_{\mathcal{M}}(n) = \max\{\mathit{chrtime}_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{VG} \wedge \mathit{inputsize}(\mathbb{G}) = n\}$$

where inputsize is a function which returns the size of a goal.



```
% horizontal:          vertical:
word(A,B,C,D,E),       word(A,F,I),
                       word(J,M,R),
word(I,J,K),           word(C,G,K,N,S),
word(M,N,O,P),         word(O,T),
word(Q,R,S,T,U),       word(E,H,L,P,U)
==> printsolution(...).
```

Figure 6.3: Solver program for some crossword puzzle.

Example 6.2.5 (time complexity of PRIMES)**.** *Consider the CHR machine*

$$\mathcal{M}_P = (\mathit{Prolog}, \mathsf{PRIMES}, \{\mathbf{upto}(n) | n \in \mathbb{N}\}),$$

where the program PRIMES *is that of Listing 1.2 on page 19. An appropriate goal size function is given by inputsize*(`upto(`n`)`) $= n$*. Every derivation starting with* `upto(`n`)` *consists of* $n-1$ **Solve** *steps (one for every computation of* `N-1`*),* $2n-1$ **Introduce** *steps (*n `upto`*/1 constraints and* $n-1$ `prime`*/1 constraints), and* $n + \mathit{nonprimes}(n) < 2n$ **Apply** *steps, where* $\mathit{nonprimes}(n)$ *is the number of composite numbers between 2 and* n *(the* `loop` *rule is applied* $n-1$ *times, the* `stop` *rule once, and the* `absorb` *rule* $\mathit{nonprimes}(n)$ *times). As a result, the time complexity* $\text{CHRTIME}_{\mathcal{M}_P}(n)$ *is smaller than* $5n-2$ *and thus it is* $O(n)$*.*

The time given by $\mathit{chrtime}_{\mathcal{M}}$ is the time needed by a *theoretical* CHR machine. The next chapter deals with the relation between this theoretical time and the time needed by *practical* CHR implementations. The distinction is important because the definition of $\mathit{chrtime}_{\mathcal{M}}$ does not correspond to the reality of CHR implementations, as the following example illustrates (cf. Theorem 11).

Example 6.2.6 (crossword puzzle)**.** *A CHR machine can solve a crossword puzzle in time linear in the number of words. Given a query containing* n *words from a dictionary as* `word`*/*k *constraints (word length* k*), the CHR machine with the program of Figure 6.3 returns all* s *solutions for a crossword puzzle in time* $O(n+s)$*. It seems highly unlikely that this time complexity can be achieved on practical computers (i.e. on a RAM machine) for arbitrary puzzles.*

Space complexity

We define the size of a CHR machine state as follows:

Definition 6.2.7 (state size function)**.**

$$\text{SIZE} : \Sigma^{\text{CHR}} \to \mathbb{N} : \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto \mathit{size}(\mathbb{G}) + \mathit{size}(\mathbb{S}) + \mathit{size}(\mathbb{B}) + \mathit{size}(\mathbb{T})$$

where for sets X*,* $\mathit{size}(X) = \sum_{x \in X} |x|$ *and the size* $|x|$ *is the usual term size.*

The space complexity of a CHR machine is defined in the usual way:

Definition 6.2.8 (space function)**.** *Given a CHR machine* $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$ *and a valid goal* $\mathbb{G}$*, the space function* $\mathit{chrspace}_{\mathcal{M}}$ *returns the largest state size encountered during the derivation for that goal:*

$$\mathit{chrspace}_{\mathcal{M}}(\mathbb{G}) = \max\{\text{SIZE}(\sigma) \mid \sigma \in \mathit{deriv}_{\mathcal{M}}(\mathbb{G})\}$$

Definition 6.2.9 (space complexity of CHR machines)**.** *Given a CHR machine* $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$*, the space complexity function* $\text{CHRSPACE}_{\mathcal{M}}$ *is defined as follows:*

$$\text{CHRSPACE}_{\mathcal{M}}(n) = \max\{\mathit{chrspace}_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{VG} \wedge \mathit{inputsize}(\mathbb{G}) = n\}$$

Example 6.2.10 (space complexity of PRIMES)**.** *Consider, as in Example 6.2.5, the CHR machine* $\mathcal{M}_P$ *and the goal size function* $\mathit{inputsize}(\texttt{upto}(n)) = n$*. If we assume all integers can be represented in a fixed number of bits, the space complexity* $\text{CHRSPACE}_{\mathcal{M}_P}$ *is* $O(n)$*. Note that if the constraint size is bounded, the size of the constraint store is asymptotically dominated by the number of* **Introduce** *steps, and the size of the built-in store is dominated by the number of* **Solve** *steps.*

6.3 Complexity-wise Completeness

In this chapter we investigate the complexity of CHR machines and their practical implementation in the Leuven CHR system. RAM machines are more realistic and faster than Turing machines, so from the complexity point of view it makes sense to focus on RAM machines. We show how to program the CHR machine to efficiently simulate RAM machines. We then discuss how to simulate CHR machines efficiently on a RAM machine, a problem that corresponds to optimizing compilation of CHR programs. This results in a general complexity meta-theorem. Finally we apply this meta-theorem to the CHR program that simulates RAM machines. This allows us to conclude that existing CHR compilation techniques, in particular those described in Chapter 3 suffice to implement the RAM machine simulator efficiently. As a result, *all* RAM machine programs (so also every known algorithm) can be translated automatically to CHR programs that have the same complexity.

6.3.1 RAM machine simulators

We now show that the CHR machine is at least as efficient as the RAM machine. Specifically, the CHR-only machine can simulate any PA-RAM machine with the same time complexity, and if $\mathcal{H}$ is a sufficiently strong host language, then CHR($\mathcal{H}$) machines can simulate standard RAM machines with the same time and space complexity.

Theorem 9. *A CHR-only machine* $\mathcal{M}_{\text{PARAM}}$ *exists which can simulate, in* $O(T+P+I)$ *time and* $O(P+J)$ *space, a* T*-time,* S*-space Peano-arithmetic RAM machine with a program of* P *lines, where* I *is the sum of the values of the input registers and* J *is the maximal sum of the values of all registers during the entire computation.*

Proof. The proof roughly corresponds to that of Theorem 8. Consider the CHR-only program PARAMSIM (see Listing 6.2), and the corresponding CHR-only machine (Φ , PARAMSIM, $\mathcal{VG}_{\text{PARAM}}$).

The mapping from CHR machine states to RAM machine states is as follows. Memory cells are represented as $\texttt{m}(A,V)$ constraints, where A is the address and V refers to the value. If V is the atom `zero`, the value is zero. Otherwise, there is a successor constraint $\texttt{s}(V,W)$ which expresses that $V = W+1$. Again, W can be either `zero` or there is another successor constraint $\texttt{s}(W,X)$, etc. For example, to represent that the register r_1 contains the value 3, the following conjunction of constraints could be used: $\texttt{m}(r_1, N_3)$, $\texttt{s}(N_3, N_2)$, $\texttt{s}(N_2, N_1)$, $\texttt{s}(N_1, \texttt{zero})$.

The mapping from a RAM machine program and input to an initial CHR goal is now obvious. We define $\mathcal{VG}_{\text{PARAM}}$ to be the image of this mapping. The RAM machine

Listing 6.2: PARAMSIM: Simulator of Peano-arithmetic RAM machines

```
i(L,L2,inc,A) \ m(A,X), c(L) <=> m(A,Z), s(Z,X), c(L2).
i(L,L2,dec,A) \ m(A,X), s(X,Y), c(L) <=> m(A,Y), c(L2).
i(L,L2,clr,A) \ m(A,X), c(L) <=> m(A,zero), c(L2).
i(L,L2,jmp,A) \ c(L) <=> c(A).
i(L,L2,cjmp,A,J), m(A,zero) \ c(L) <=> c(J).
i(L,L2,cjmp,A,J), m(A,X), s(X,_) \ c(L) <=> c(L2).
i(L,L2,halt) \ c(L) <=> true.
```

program is encoded as $\texttt{i}/\{3,4,5\}$ constraints; the first argument represents the label (or line number), the second argument is the label of the next program line, the other arguments represent the actual instruction. The input memory cells are represented as above. The program counter is set to the label of the first line of the program by adding a corresponding initial $\texttt{c}/1$ constraint.

After the initialization, one rule is applied for every step of the RAM machine. Applying a rule causes at most three constraints to be inserted. Therefore, the number of CHR steps is bounded by four times the number of RAM machine steps, plus P steps to **Introduce** the $\texttt{i}/\{3,4,5\}$ constraints. Initializing the input memory cells to their values means inserting one $\texttt{m}/2$ constraint for every input and I $\texttt{s}/2$ constraints. The number of registers (and hence $\texttt{m}/2$ constraints) is bounded by the program size P (remember we don't have indirection), so the total time complexity of the CHR machine is $O(T+P+I)$. The CHR store contains P $\texttt{i}/\{3,4,5\}$ constraints; since the number of $\texttt{m}/2$ constraints is $O(P)$ and the number of $\texttt{s}/2$ constraints is bounded by J, the space complexity of the CHR machine is $O(P+J)$. □

The PARAMSIM program does not use any host language arithmetic operations. Its unary number representation causes an exponential space penalty. If host language arithmetic is available, a similar program can be written which uses only $O(P+S)$ space by directly storing the value of a register in the second argument of the $\texttt{m}/2$ constraints.

We can also simulate the more realistic standard-arithmetic RAM machine in CHR. However, if we want to do this without a harsh complexity penalty, we need host language support to handle the integers.

Theorem 10. *For any sufficiently strong host language $\mathcal{H}$, a CHR($\mathcal{H}$) machine $\mathcal{M}_{\mathrm{RAM}}$ exists which can simulate, in $O(T+P+S)$ time and $O(S+P)$ space, a T-time, S-space standard RAM machine with a program of P lines.*

Proof. As in Theorem 9, for the simulator program of Listing 6.3. The RAM memory representation is simpler since integer numbers are available. The representation of a RAM machine program and memory is as before, except that we assume input registers to be in a continuous range $0,\ldots,m$ and a constraint $\texttt{maxm}(m)$ is added to the initial CHR goal. The auxiliary constraint $\texttt{maxm}(m)$ indicates that the current highest initialized register address is m. When a register with a higher address n is initialized, the auxiliary constraint $\texttt{initm}/3$ is used to initialize all addresses in the range $m+1,\ldots,n$. In addition to the $O(P)$ time to **Introduce** the encoded program and the $O(T)$ time to simulate it (again one CHR rule application per program instruction), the simulator needs $O(S)$ time to initialize memory. □

Listing 6.3: RAMSIMUL: Simulator of standard RAM machines

```
i(L,init,A), m(A,B), maxm(M) \ c(L) <=> initm(M+1,B,L).
  initm(A,B,L) <=> A =< B | m(A,0), initm(A+1,B,L).
  initm(A,B,L), m(B,X) <=> A > B | m(B,0), maxm(B), c(L+1).
i(L,cnst,B,A) \ m(A,X), c(L) <=> m(A,B), c(L+1).
i(L,add,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X+Y), c(L+1).
i(L,sub,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X-Y), c(L+1).
i(L,mul,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X*Y), c(L+1).
i(L,div,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X//Y), c(L+1).
i(L,mov,B,A), m(B,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,imv,B,A), m(B,C), m(C,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,mvi,B,A), m(B,Y), m(A,C) \ m(C,_), c(L) <=> m(C,Y), c(L+1).
i(L,jmp,A) \ c(L) <=> c(A).
i(L,cjmp,A,J), m(A,0) \ c(L) <=> c(J).
i(L,cjmp,A,J), m(A,X) \ c(L) <=> X =\= 0 | c(L+1).
i(L,halt) \ c(L) <=> true.
```

Note that for a fixed RAM machine program which uses at least as much time as space, P is a constant and S is $O(T)$, so the CHR simulator RAMSIMUL takes $O(T)$ time and $O(S)$ space.

6.3.2 Complexity meta-theorem

The previous section dealt with the complexity properties of the theoretical CHR machine. In this section we investigate the complexity achievable in practice, i.e. on a RAM machine, which corresponds (more or less) to a real computer. As a result, we get a complexity meta-theorem; in Section 6.3.3 we will then apply this meta-theorem to the RAM machine simulator program RAMSIMUL. This will allow us to conclude that every algorithm can be implemented efficiently in CHR.

CHR is Turing-equivalent

In Chapter 2 we have discussed the compilation of CHR. Most of the CHR compilers translate CHR programs to host language programs. The resulting host language code can be executed on RAM machines by an interpreter or via further compilation steps. In this sense, we can say that CHR compilers convert CHR programs to (RAM machine) executable code. So, because clearly CHR compilers do exist, RAM machines can simulate CHR machines:

Lemma 6.3.1. *The RAM machine can simulate CHR machines.*

Because CHR machines can simulate Turing machines (Theorem 8), and Turing machines can simulate CHR machines (Lemma 6.1.9 and the above), we get the following rather unsurprising result:

Corollary 6.3.2. *The CHR machine is Turing-equivalent.*

Complexity of the compiled code

We now examine the practically achievable complexity of simulating a CHR machine on a RAM machine. We consider existing CHR compilers, in particular, the Leuven CHR

system in hProlog (see Section 1.3.4).

First we recapture some compiler optimizations that are crucial for the time and space complexity of the generated code. We then define the *dependency rank* of a constraint occurrence, which will turn out to play a crucial role in the time complexity of executing a CHR machine.

The refined operational semantics. Most CHR implementations — in particular, the Leuven CHR system — follow the refined operational semantics ω_r of CHR. The ω_r semantics is discussed in detail in Chapter 2.

Essential in the ω_r semantics is the notion of an *active* constraint. Query and body constraints are introduced from left to right. Once a constraint is introduced (or triggered by a **Solve** transition), it becomes active: its occurrences in the program are tried, in textual order. For every occurrence, the corresponding rule is tried by looking up matching *partner* constraints.

Join ordering. As discussed in [Sneyers, 2008b], chapter 9, the order in which partner constraints are looked up is important. CHR compilers implement a strategy to pick the order of partner constraint lookups, called the *join ordering*. Given a CHR program $\mathcal{P}$, a join ordering strategy $\prec$ induces for every head constraint occurrence c of $\mathcal{P}$, an order $\prec_c^{\mathcal{P}}$ on its partners.

Functional dependencies. In [Sneyers, 2008b, Chapter 4] the notion of functional dependencies between constraint arguments was introduced [Duck and Schrijvers, 2005]. Informally, for a given CHR store, a constraint has a *set semantics* functional dependency on certain key arguments if there are no two instances of the constraint with the same key arguments.

In the refined operational semantics, set semantics functional dependencies can easily be enforced using simpagation rules. For example, to make sure that `c/4` has a functional dependency on the combination of its first and third argument, we add the following rule before all other rules:

```
c(A,_,B,_) \ c(A,_,B,_) <=> true.
```

Indexing. As we have seen in Chapter 3, advanced data structures can be used to implement the CHR constraint store in an efficient way. The join ordering strategy determines which combinations of key arguments (look-up patterns) are used for partner constraint look-ups. The constraint store is implemented in such a way that for every combination of constraint arguments that is used as a look-up pattern, an index is maintained, for instance using hash-tables. As a result, all constraint store operations can be done in $O(1)$ amortized time. Furthermore, in terms of space, these data structures have an overhead of only a constant factor.

Determined partners and dependency rank. We now introduce the notions of determined partners and dependency rank. This will allow us to get a tighter upper bound on the complexity of finding suitable partner constraints.

Definition 6.3.3 (determined partner)**.** *Given a join ordering strategy $\prec$, a CHR program $\mathcal{P}$, and a set of valid goals $\mathcal{VG}$, we say an occurrence c is* determined *by the j-th occurrence of constraint a iff for all execution states σ that occur in a derivation $d \in \Delta_{\omega_r}^{\mathcal{H}}|_{\mathbb{G}}$ for some*

valid goal $\mathbb{G} \in \mathcal{WG}$*, the following holds: if* σ *is of the form* $\langle [a\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ *(that is, the occurrence subprocedure for the* j*-th occurrence of constraint* a *is about to be executed), then a set semantic functional dependency for* c *holds in state* σ*, where the key arguments of* c *are fixed by* a *and all partners* x *for which* $x \prec_a^{\mathcal{P}} c$*.*

In other words, a partner constraint c is determined by a given (active) constraint occurrence of a if the following holds: whenever the partner constraint c is looked up in the corresponding occurrence subprocedure for a, there is at most one match for c that needs to be considered.

Definition 6.3.4 (dependency rank)**.** *The dependency rank of an (active) occurrence* a *is the number of non-determined partner constraints of* a*.*

The following example illustrates the above definitions. Consider the join ordering strategy $\prec$ used in the Leuven CHR system, the program RAMSIMUL, and the set of valid goals corresponding to the valid RAM machine instances.

Example 6.3.5 (dependency rank)**.** *Consider the fifth rule of* RAMSIMUL*:*

```
i(L,add,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X+Y), c(L+1).
```

For the active occurrence `c(L)`*, the following join ordering is used:*

$$\texttt{c(L)} \prec^{\text{RAMSIMUL}}_{\texttt{c}(L)} \texttt{i(L,add,B,A)} \prec^{\text{RAMSIMUL}}_{\texttt{c}(L)} \texttt{m(B,Y)} \prec^{\text{RAMSIMUL}}_{\texttt{c}(L)} \texttt{m(A,X)}.$$

Given the first argument L *there can be only one matching* `i/4` *constraint, which means the first argument of each* `m/2` *constraint is known. Again, for a given first argument, there can be only one* `m/2`*. These functional dependencies are not enforced explicitly by simpagation rules; they are implied by the set of valid goals and the rules. Since all its partners are determined, the dependency rank of* `c(L)` *is zero.*

Similarly, the other occurrences of `c/1` also have dependency rank zero. In general, the dependency rank is of course not always zero. Different occurrences in the same rule may even have a different dependency rank.

Example 6.3.6 (non-zero dependency ranks)**.** *Consider the following rule:*

```
m(A,B), m(B,C), m(D,E), m(E,F), m(G,H) <=> ...
```

Assume that `m/2` *constraints are uniquely determined by their first argument (as in the* RAMSIMUL *program) and that the join ordering is the textual order. The dependency rank of* `m(A,B)` *is two:* `m(D,E)` *and* `m(G,H)` *contribute to it while the other head constraints are determined partners. The dependency rank of* `m(B,C)` *is three:* `m(D,E)`*,* `m(G,H)`*, and* `m(A,B)` *contribute to it.*

Space reuse. Every time a CHR constraint is removed, its representation in memory becomes garbage. If this garbage is not collected, we may get a space complexity which is not linear in the size of the constraint store. Using garbage collection we can get the right space complexity. However, in most Prolog systems, garbage collection has a time complexity linear in the number of live cells. This may result in a severe time complexity penalty.

In [Sneyers et al., 2006] we have tackled this problem by introducing memory reuse techniques called *in-place updates* and *suspension reuse*, inspired by compile-time garbage

collection [Mazur, 2004]. These optimizations improve the space complexity by eliminating garbage, with only a small constant factor worst-case time overhead. The basic idea of suspension reuse is to store the representation of a removed constraint in a cache. Later, when a new constraint has to be inserted, a representation from the cache is used to build the new constraint representation. In-place updates are a special case where both the removal and the insertion are in the same rule, eliminating the need for an intermediate cache.

Lemma 6.3.7. *Using suspension reuse with unlimited cache size, the following holds: If during a particular execution, the maximal number of constraints in the store is S_{max}, then at any point in the execution, $S + C \leq S_{max}$, where S is the number of constraints in the store and C is the number of elements in the cache.*

Proof. Execution consists of a sequence of insertion and removal operations. We denote the store size after the i-th operation with S_i, the cache size with C_i, and their sum with $M_i = S_i + C_i$. Initially both the store and the cache are empty: $S_0 = C_0 = 0$. Insertion increments the store size and decrements the cache size if it is not already empty (otherwise it simply remains empty). Removal decrements the store size and increments the cache size. We proceed by induction on the sequence length. For zero-length sequences the property holds trivially. Assuming the property holds for any sequence of length n, we show that it also holds for sequences of length $n+1$. Because of the induction hypothesis:

$$\forall x \leq n: \quad M_x \;\leq\; \max_{i \leq n} S_i \quad \leq \quad \max_{i \leq n+1} S_i \;=\; S_{max}$$

We now only have to show that $M_{n+1} \leq S_{max}$. If the last operation in the sequence is a removal, then $M_{n+1} = (S_n - 1) + (C_n + 1) = M_n \leq S_{max}$. Assuming the last operation is an insertion, there are two cases: if $C_n > 0$, then $M_{n+1} = (S_n + 1) + (C_n - 1) = M_n \leq S_{max}$; otherwise $C_n = 0$ and $M_{n+1} = S_{n+1} \leq S_{max}$. This concludes the proof. □

With an unlimited cache size, no constraint representation ever becomes garbage (from the point of view of the underlying host language in which the CHR system is implemented): all constraint representations are alive, either because they are in the store or because they are in the cache. The above lemma shows that the right space complexity can be achieved for all CHR programs — without having to resort to run-time garbage collection in the host language.

Complexity meta-theorem

We now state the main result of this section:

Theorem 11. *Given a CHR program $\mathcal{P}$ and a ω_t derivation d of length T which has a corresponding ω_r derivation, for which the maximal store size is S, m is the maximum dependency rank of the active occurrences in $\mathcal{P}$, and p is the number of propagation rule applications in d; assuming the host language constraints used in the guards and bodies of the rules of $\mathcal{P}$ can be evaluated in constant time; the Leuven CHR system compiles $\mathcal{P}$ to hProlog code which has, for the given derivation d, a time complexity $O(TS^{m+1})$ and a space complexity $O(S + p)$.*

Proof. Assume the derivation d consists of s **Solve** steps, i **Introduce** steps, and a **Apply** steps: $T = s+i+a$. The cost of finding a rule match for an active occurrence is $O(S^m)$ since this process basically boils down to nested iteration over the constraints in the store, where the nesting depth is the dependency rank. Indeed, determined partners only contribute a

constant factor to this cost. Checking and extending the propagation history can be done in constant time if the history is implemented as a hash table. The **Apply** steps take $O(a)$ time plus the time to find rule matches. The latter has to be taken into account even if no rule matches are found and no rule is applied, so we attribute it to the **Introduce** and **Solve** steps. The **Introduce** steps take $O(iS^m)$ time: for every active occurrence of the introduced constraint, a matching rule is sought; for a fixed CHR program, the number of occurrences is bound by a constant. Every **Solve** step potentially triggers all the $O(S)$ constraints in the store, so the **Solve** steps may take up to $O(sS^{m+1})$ time. Since s, i, and a are all $O(T)$, the total time complexity is $O(TS^{m+1})$. The $O(S+p)$ space complexity can be achieved using suspension reuse with unlimited cache size, as shown in Lemma 6.3.7. □

Theorem 12. *If in the previous theorem, the CHR program is ground (i.e. all constraint arguments are ground), then $O(TS^m)$ time complexity can be achieved.*

Proof. See the previous proof. Constraints are never triggered in ground programs. This reduces the complexity of one **Solve** step to a constant. □

Formulating the above result in terms of CHR machines we get:

Corollary 6.3.8. *A ground CHR machine without propagation rules, with time complexity T and space complexity S, can be simulated on a RAM machine with time complexity $O(TS^m)$ and space complexity $O(S)$, where m is the maximum dependency rank of the active occurrences in the program of the CHR machine.*

6.3.3 Complexity-wise completeness

Now we use the general result of Corollary 6.3.8 to analyze the time and space complexity of the RAM simulation (compiler-generated code) of the CHR machine $\mathcal{M}_{\text{RAM}}$, which itself simulates RAM machines.

Theorem 13. *The CHR machine $\mathcal{M}_{\text{RAM}}$ with program* RAMSIMUL*, with time complexity T and space complexity S can be simulated on a RAM machine with time complexity $O(T)$ and space complexity $O(S)$.*

Proof. It can be easily verified that the dependency rank for all occurrences of `c/1` is zero given the join ordering strategy used in the Leuven CHR system (cf. Example 6.3.5). For the other occurrences this is slightly less straightforward. Valid goals have exactly one `c/1` constraint. All rules remove one `c/1` constraint and optionally insert another one in the body (indirectly in the case of the rule for the `init` instruction). Hence, there is never more than one `c/1` constraint — in other words, `c/1` has a set semantics functional dependency on the empty key. If the join order strategy does the lookup of `c/1` first, the remaining partners become determined.

Since the dependency rank is zero for all occurrences, applying Corollary 6.3.8 (with $m = 0$) results in the desired complexities. □

We conclude that "everything can be done efficiently in CHR":

Corollary 6.3.9. *For every (RAM machine) algorithm which uses at least as much time as it uses space, a CHR program exists which can be executed in the Leuven CHR system in hProlog, with time and space complexity within a constant from the original complexities.*

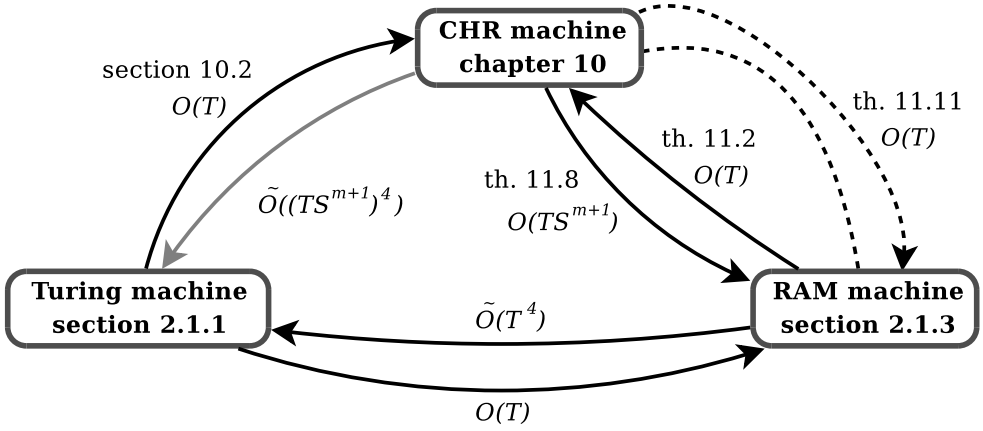


Figure 6.4: Relationships between Turing, RAM, and CHR machines

Proof. Consider any algorithm which can be expressed as a RAM machine program with a program of P lines, and let its time and space complexities be T and S, respectively. Because of Theorem 10, a CHR(Prolog) machine $\mathcal{M}_{\text{RAM}}$ with program RAMSIMUL exists, which simulates that RAM machine in $O(S)$ space and $O(T)$ time, since P is a constant and S is $O(T)$. Now, because of the above Theorem 13, executing RAMSIMUL in the Leuven CHR system also takes $O(T)$ time and $O(S)$ space. □

One may expect to pay *some* performance penalty for using a very high-level language like CHR, so it is comforting that at least we can always get the asymptotic complexities right.

Discussion

We have investigated the complexity of Constraint Handling Rules by introducing the CHR machine, a model of computation based on the operational semantics of CHR. Besides the expected result that CHR is a Turing-equivalent language, we have demonstrated the much stronger result that every RAM machine program can be implemented as a CHR program which has the same asymptotic time and space complexities if executed in the Leuven CHR system. In other words, the current state-of-the-art in CHR compilation allows CHR programmers to implement any algorithm with the best possible complexity. As far as we know, CHR is the first declarative language for which such a complexity-wise completeness result has been proven to hold within the pure part of the language, i.e. without imperative extensions to the language — see also Section 6.3.4.

Figure 6.4 summarizes the results. It gives an overview of the relationships between three models of computation: Turing machines, RAM machines, and CHR machines. An arrow between A and B indicates that A can be simulated on B. Labels indicate the relevant section or theorem and the time complexity of simulating a T-time, S-space A on B. Note that the $\tilde{O}((TS^{m+1})^4)$ bound for simulating a CHR machine on a Turing machine can be obtained by simulating the CHR machine on a RAM machine, and then simulating that RAM machine on a Turing machine.

Listing 6.4: Alternative register initialization, using the ω_r semantics

```
prog(L,init,A), m(A,B), m(B,_) \ pc(L) <=> pc(L+1).
prog(L,init,A), m(A,B)         \ pc(L) <=> m(B,0), pc(L+1).
```

Register initialization

Our definition of the space complexity of a RAM machine (see Definition 6.1.8 on page 166) is based on that of Savitch [1978]. It counts all registers in the used address range, whether or not each individual register was effectively used. In the literature, other definitions of RAM machine space complexity only take the used registers into account [van Emde Boas, 1990]. If we would use such a definition, the above results no longer hold in general: for a program that uses the registers in a sparse way, the RAM machine simulator RAMSIMUL of Listing 6.3 would use more space (and thus possibly also more time) than the original RAM machine. The reason is that the simulator initializes the entire register range.

However, by relying on the refined operational semantics, we can implement the `init` instruction in a different way — see Listing 6.4. The auxiliary constraints `maxm`/1 and `initm`/3 are not needed in this version of the simulator. Of course, this alternative program is no longer confluent; its correctness depends on the order of rule applications enforced by the refined semantics. Indeed, under the ω_r semantics the second rule is applied only if the first rule cannot be applied because of the absence of a corresponding `m`/2 constraint. Checking for absence of constraints cannot be done in a confluent way — see also the work on extending CHR with negation-as-absence (cf. Section 1.6.2).

Related and future work

We have already mentioned the complexity meta-theorem by Frühwirth 2002b, 2002a in Section 1.2. Since it assumes a very naive implementation of CHR, the resulting bounds are rather crude. For the RAM machine simulator program RAMSIMUL simulating a T-time RAM machine, the upper bound predicted by Frühwirth [2002b] is $O(T^6)$ — quite far from the $O(T)$ bound of Lemma 13.

We have explicitly decoupled the two steps in the approach of Frühwirth by introducing CHR machines. If suitable termination orders can be found, they can be used to show an upper bound on the complexity of the CHR machine. This is the first step. However, for programs that are non-terminating in general, like the RAM simulator, or for which no ranking can be found, other techniques have to be used to prove complexity properties. For example, we have shown that the steps of the CHR machine $\mathcal{M}_{\text{RAM}}$ correspond to the steps of the RAM machine it simulates, terminating or not. The second step corresponds to the question of how efficiently a CHR machine can be executed in practice. Recent work on optimizing compilation of CHR has allowed us to achieve much tighter bounds.

Future work. It is an open problem whether a result similar to the the linear speedup theorem [Hartmanis and Stearns, 1965] can be demonstrated for CHR machines. To improve the time complexity of a CHR machine, one could try to reduce the number of **Apply** steps by combining rules, and the number of **Introduce** steps by combining constraints. It is not clear whether such a reduction is possible in general. This is somewhat related to partial evaluation techniques.

Although we are convinced that every algorithm can be implemented with an *elegant* CHR program, it remains a useful research topic to construct good CHR implementations

of existing (or new!) algorithms (cf. [Sneyers, 2008b], chapter 5).

The RAM machine simulator is a ground CHR program without propagation rules. In a sense, our result implies that non-ground constraints (which may be triggered) and propagation rules (that require checking and maintaining a propagation history) are not strictly needed for complexity-wise completeness. However, since non-ground constraints and propagation rules are widely used (especially in the traditional constraint solver programs), improving the complexity of their implementation is still very useful.

6.3.4 Other declarative languages

Many Turing-complete programming languages have the complexity-wise completeness property we have shown for X = CHR(hProlog): "every algorithm can be implemented in language X with the right time and space complexity". For instance, in all imperative languages that we are aware of, it is a straightforward exercise to construct a RAM machine simulator and show that it has the right complexity. After all, the basic ingredients needed for a RAM machine simulator are directly available in most imperative languages.

However, for higher level, declarative languages, the complexity-wise completeness property is far less trivial. The time and space complexity of a program depends more crucially on optimizing compilation.

In this section we briefly investigate whether some other declarative languages allow an efficient implementation of a RAM machine simulator, given the current state of the art. To keep this section concise, we will only consider one well-known (and supposedly representative) language for each of these declarative paradigms: logic programming (Prolog), functional programming (Haskell), term-rewriting (Maude), and rule-based programming (Jess).

Whether or not the complexity-wise completeness result holds for some language clearly depends largely on the properties of its compiler. A pathological compiler can be conceived that detects the pattern of a 'RAM machine simulator program' as a special case and produces special, hardwired output with the desired complexity properties. Such 'optimizations' have to be ruled out since they are not applicable to a large class of programs. However, where exactly to draw the line between 'cheating' and 'benign' optimizations is not straightforward. In this section we only consider the 'pure' fragments of the different declarative languages. Some of the languages also have imperative extensions (e.g. `setarg/3` in Prolog) but we consider those a form of 'cheating' — for the CHR(Prolog) complexity-wise completeness result we did not need any extensions of CHR, and the compiler optimizations we needed are 'benign' in the sense that they are applicable to a large class of CHR programs.

Sufficient ingredients

By closely investigating the definition of RAM machines (see Section 6.1.1), we can identify the crucial programming language features needed to obtain a complexity-wise completeness result in the same way as in the previous section, that is, by constructing a RAM machine simulator. An efficient RAM machine simulator can be implemented if the following ingredients can be implemented:

1. A mechanism for iteration, such that iterating n times takes $O(nT_s)$ time and $O(S_s)$ space if evaluating the stop condition takes T_s time and S_s space;

2. The arithmetic operations, with the same complexity as the corresponding RAM machine arithmetic operations;

3. An *if-then-else* language construct and evaluation of (syntactic) equality and inequality conditions, both in constant time and zero space;

4. Dynamically growing arrays which allow n insertion, g lookup, and s update operations in $O(n + g + s)$ time and $O(n)$ space.

Most declarative languages do not offer iteration (the first ingredient) as a basic language construct, but many implementations convert tail recursion to iteration, so the first ingredient will not be the problem. The second and third ingredient are directly and/or implicitly available in all the languages we consider.

Arrays in declarative languages

The remaining ingredient, an efficient dynamically growing array, is the one that seems to be the most difficult to implement. In this section we try to implement, as efficiently as possible, the functionality of arrays in several declarative languages, and we construct corresponding RAM machine simulator programs. In the next section we will experimentally compare the performance of the resulting programs.

Logic programming languages. We use the term *pure Prolog* to denote the Prolog language as described by Clocksin and Mellish [1984], without the *assert* and *retract* built-ins. Clearly, if non-pure Prolog extensions — for example global variables, mutable terms, and assert/retract — are allowed, there is an efficient RAM machine simulator implementation: consider, for instance, the code the CHR(hProlog) compiler generates for the RAMSIMUL program.

To the extent of our knowledge, there is no Prolog system which allows an efficient pure Prolog implementation of dynamically growing arrays. Association lists, available in many Prolog systems as a standard module called `assoc`, can be used instead. The implementation of `assoc` that is used in hProlog is based on an implementation by Mats Carlsson (which was based on an implementation by Richard O'Keefe) based on AVL-trees [Adelson-Velsky and Landis, 1962]. Lookup, insertion and update take $O(\log n)$ worst-case time. Listing A.3 in [Sneyers, 2008b] gives a Prolog program which uses association lists to implement a RAM simulator. Association lists are used to represent both the RAM machine program and the memory cells.

Mercury. Mercury [Somogyi et al., 1996] is a strongly-typed, high-performance logic programming language. The Mercury system includes an `array` module. However, the procedures of this module are not written in the Mercury language itself, but directly in the target languages (C, C#, and Java).

In an experimental development branch of Mercury, compile-time garbage collection (CTGC) has been added by Mazur [2004]. This allows automatic *structure reuse* in a large class of Mercury programs. Perhaps CTGC allows a reasonably efficient pure Mercury implementation of growing arrays: using AVL-trees and with in-place updates thanks to CTGC, it should be possible to perform n insertions, g lookups, and s updates in $O((n+g+s)\log n)$ time and $O(n)$ space. We have not been able to test Mercury programs experimentally since CTGC is not yet available in the main release of the Mercury system.

Functional programming languages. Haskell [Hudak et al., 2007] is a modern typed, lazy, purely functional language. Most Haskell systems include the `Data.Array` module

in their standard libraries. This module efficiently implements arrays, but it is not implemented in Haskell itself. The fastest pure Haskell implementation of arrays we could find is available in the standard `Data.IntMap` module, which is based on an implementation of Patricia trees [Morrison, 1968] by Okasaki and Gill [1998]. This data structure allows memory cell look-ups, updates, and insertion (initialization) in $O(\min(n, W))$ time, where W is the number of bits in an `Int`. On our test platform, $W = 32$ so the operations can be considered to be constant-time. However, since the updates are not done in-place, the space complexity is $O(n + s)$ instead of $O(n)$. Listing A.4 in [Sneyers, 2008b] gives the Haskell program we have tested. We have tested both a "lazy" version and a "strict" version. The latter naively forces all lazy thunks immediately to weak head normal form (WHNF); it differs from the lazy version on two accounts only. Firstly, the fields of the `Instr` datatype are declared to be strict. Secondly, each function application `f e` is transformed into `let x=e in x ‘seq‘ f x`, which forces the subexpression `e` to WHNF before evaluating the main expression `f e`.

Term-rewrite systems. Maude[4] [Clavel et al., 2002] is a system for declarative programming in rewriting logic. It features efficient rewriting of terms with associative-commutative (AC) operators using the *stripper-collector matching* algorithm of Eker [2003]. Listing A.5 in [Sneyers, 2008b] gives the Maude program we have tested. This program is directly derived from the CHR rules. As in CHR, the collection data structures and the operations on them are implicit. In this sense CHR and Maude are higher level languages than Prolog and Haskell.

Unfortunately, for the rules of Listing A.5 in [Sneyers, 2008b], the current implementation of Maude is not able to use its most efficient matching algorithm. By making the data structure operations more explicit (using the `Map{Int,Int}` module) we obtain a more efficient program in which memory cell lookups take only logarithmic time. It is given in Listing A.6 in [Sneyers, 2008b].

Rule engines. Jess[5] [Friedman-Hill, 2003] is considered to be one of the fastest rule engines. Like its ancestor CLIPS [Giarratano and Riley, 1994], it uses the RETE algorithm of Forgy [1982]. Listing A.7 in [Sneyers, 2008b] gives the Jess program we have tested.

In a sense, Jess is higher level than Prolog and Haskell because the data structures are implicit. It is lower level than CHR and Maude (as in the Maude program of Listing A.5 in [Sneyers, 2008b]) because the data structure *operations* (`assert`, `retract`, and `modify`) are explicit. Moreover, as far as we know, Jess does not have the join ordering optimization. We have picked the best possible order of rule heads in the simulator program of Listing A.7 in [Sneyers, 2008b]. If the heads are written in a different order, performance suffers. This is another sense in which Jess can be considered to be a lower level language than CHR — thanks to automatic join ordering, CHR programmers do not have to worry about the order of the head constraints.

Experimental results

To test the performance of the RAM machine simulator, we have executed the simulator for three benchmark RAM machine programs (see Table 6.3). The results are shown in Table 6.4, which lists the execution times of running the RAM simulator benchmarks of Table 6.3 in different RAM simulator implementations.

[4] Maude home page: `http://maude.cs.uiuc.edu/`
[5] Jess home page: `http://www.jessrules.com/`

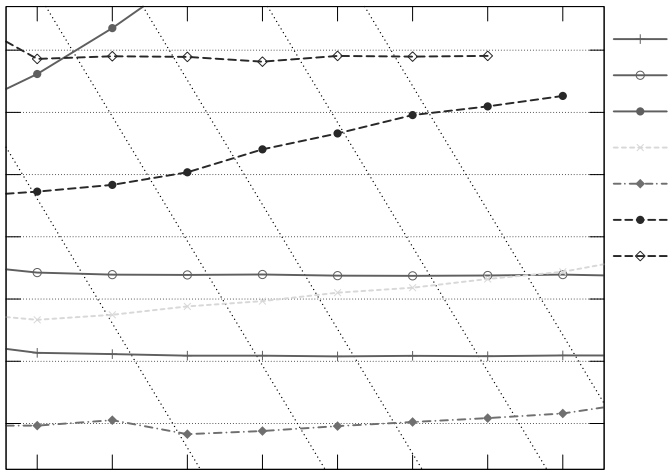


Figure 6.5: Results of the "NLoop" benchmark (cf. Table 6.4)

Benchmark 1: "Loop". The first benchmark, "Loop", performs $O(n)$ operations and uses only three memory cells. The space usage is constant in CHR, Prolog, Maude, and strict Haskell, but not in lazy Haskell: in this example, lazy evaluation creates $O(n)$ lazy thunks so it needs $O(n)$ space. The time complexity is linear in all systems. The Jess program is the slowest: it takes more than twice the time of the naive Maude program. The naive Maude program is roughly three times slower than the one that uses `Map`, which is about as fast as CHR. The CHR program with type detail[6] level 2 is almost four times faster than the CHR programs with type detail level 1 or 0, about twice as slow as the Prolog version and four times slower than strict Haskell.

Benchmark 2: "MFib". In the second benchmark, "MFib", $O(n)$ memory cells are used. Naive Maude and CHR without type information do not get the time complexity right: the $O(n)$ lookups seem to take $\tilde{O}(n^2)$ time. Prolog, Haskell, and Maude with `Map` get the time complexity almost right: $\tilde{O}(n)$ instead of $O(n)$. Both Prolog and Haskell use too much space. In the case of Prolog and strict Haskell, garbage collection takes an increasingly higher proportion of the runtime. In lazy Haskell, even garbage collection does not prevent running out of memory. CHR with type detail level 1 and Jess get the right time and space complexity; Jess is more than ten times slower and also uses significantly more space.

Benchmark 3: "NLoop". The last benchmark, "NLoop", performs $O(n^2)$ updates on $O(n)$ memory cells. As expected from the results of the previous benchmark, only CHR with type detail level > 0 and Jess get the $O(n^2)$ time complexity completely right. CHR is between 10 and 30 times faster than Jess. Prolog, (strict) Haskell and Maude with `Map` achieve $\tilde{O}(n^2)$ time complexity.

To aid us in interpreting the benchmark results, we have plotted the runtimes divided by the expected complexity $O(n^2)$ in Figure 6.5. If the resulting curve is horizontal, the

[6] See Table 7.1 in [Sneyers, 2008b]. Type detail level 2 corresponds to types and modes (crucially, `+dense_int` for the first argument of `m`/2); detail level 1 corresponds to modes only (all ground); detail level 0 corresponds to no argument information.

(a) Loop

```
i(1,add,1,3)
i(2,sub,1,2)
i(3,cjmp,2,5)
i(4,jmp,1)
i(5,halt)

m(1,1)
m(2,n)
m(3,0)
c(1)
```

(b) MFib

```
i(1,init,3)
i(2,imv,1,6)
i(3,imv,2,7)        m(1,8)
i(4,mul,6,7)        m(2,9)
i(5,mvi,7,3)        m(3,10)
i(6,add,5,1)        m(4,n)
i(7,add,5,2)        m(5,1)
i(8,add,5,3)        m(6,0)
i(9,sub,5,4)        m(7,0)
i(10,cjmp,4,12)     m(8,1)
i(11,jmp,1)         m(9,1)
i(12,halt)          c(1)
```

(c) NLoop

```
i(1,mov,2,3)        i(16,mov,3,5)
i(2,add,1,3)        i(17,sub,1,5)
i(3,init,3)         i(18,cjmp,5,20)
i(4,mvi,4,3)        i(19,jmp,12)
i(5,sub,4,3)        i(20,sub,4,2)
i(6,mov,3,5)        i(21,cjmp,2,23)
i(7,sub,1,5)        i(22,jmp,10)
i(8,cjmp,5,10)      i(23,halt)
i(9,jmp,3)
i(10,mov,2,3)       m(1,10)
i(11,add,1,3)       m(2,n)
i(12,imv,3,5)       m(3,0)
i(13,add,4,5)       m(4,1)
i(14,mvi,5,3)       m(5,0)
i(15,sub,4,3)       c(1)
```

Table 6.3: Example RAM simulator queries used for benchmarking

expected complexity is achieved. Both in constant factors and unwanted non-constant factors, the strict Haskell version is better than the Prolog version, which is in turn better than the Maude version.

Summary

We can classify the languages as shown in Table 6.5:

- Jess and CHR (with modes) achieve optimal time and space complexity;
- Maude with `Map` has optimal space complexity (as far as we can tell) and gets within a (large) polylogarithmic factor from optimal time complexity;
- CHR without type declarations and naive Maude have optimal space complexity, but they do not achieve optimal time complexity;
- Prolog and strict Haskell have a time complexity which is within a polylogarithmic factor from optimal, but their space complexity is not optimal;
- lazy Haskell does not get close to optimal space complexity, and this ruins its time complexity.

For the declarative programming languages we have tested in this section, the apparent inability to obtain optimal complexity in the pure language is not really a problem in practice. If needed, programmers can use non-pure language elements like the `setarg/3` built-in in Prolog and the `array` modules in Mercury and Haskell. It is not clear to us whether and how pure logic or functional programming languages can be implemented in a way that allows an efficient implementation of a RAM simulator.

6.3.5 Constant factors

We have shown the complexity-wise completeness of CHR — every algorithm can be implemented in CHR with the right asymptotic time and space complexity. However, the constant factors hidden behind the notion of asymptotic complexities could be huge. In fact, they could be so huge as to be completely paralyzing in practice. In this section we investigate these constants experimentally.

Query	CHR(Prolog)			Prolog	Haskell		Maude		Jess
n	2	1	0		lazy	strict	naive	Map	
Loop									
10^4	0.09	0.32	0.32	0.05	0.05	0.02	1.01	0.37	3.48
10^5	0.88	3.23	3.10	0.44	0.41	0.21	10.09	3.48	24.07
10^6	8.70	32.42	29.93	4.46	5.40	2.05	100.43	35.32	231.13
10^7	86.95	324.91	298.11	42.88	mem	20.41	987.28	371.54	2266.33
MFib									
10^3	0.03	0.07	2.20	0.05	0.04	0.02	32.00	0.24	3.08
10^4	0.26	0.69	434.09	0.60	1.19	0.16	4772.70	3.69	10.22
10^5	2.61	7.39	—	7.40	mem	1.82	—	52.12	98.72
10^6	26.76	76.43		89.37		19.69		1650.49	mem
NLoop									
2^7	0.18	0.44	4.02	0.26	0.17	0.08	57.05	1.08	4.75
2^8	0.71	1.72	26.81	1.10	0.73	0.34	419.54	4.68	19.59
2^9	2.79	6.86	192.55	4.83	9.43	1.16	3677.66	21.51	77.81
2^10	11.16	27.57	1475.77	20.49	mem	4.82	—	111.13	295.20
2^11	44.28	108.87	11620.42	90.04		20.37		530.60	1258.71
2^12	178.22	434.48	—	380.78		85.27		2604.68	4997.12
2^13	709.89	1744.55		1678.73		356.29		11474.24	20147.51
2^14	2864.56	7050.31		7272.05		1500.36		51573.71	

Table 6.4: Benchmarks for several RAM simulator implementations

Language	*Time*	*Space*	*Complexity-wise complete?*
CHR [0]	not optimal	optimal	space-only
CHR [1]	optimal	optimal	yes
CHR [2]	optimal	optimal	yes
Prolog	almost optimal	not optimal	almost (time-only)
Haskell (strict)	almost optimal	not optimal	almost (time-only)
Haskell (lazy)	not optimal	not optimal	no
Maude (naive)	not optimal	optimal	space-only
Maude (`Map`)	almost optimal	optimal	almost
Jess	optimal	optimal	yes

Table 6.5: Language classification in terms of complexity-wise completeness

```
     movl    $1, %eax
     movl    $1000000, %ecx
     movl    $0, %edx
.L1: addl    %eax, %edx
     subl    %eax, %ecx
     je      .L5
     jmp     .L1
.L5:
```

```
m(1,1),
m(2,1000000),
m(3,0),
i(1, add, 1, 3),
i(2, sub, 1, 2),
i(3, cjmp, 2, 5),
i(4, jmp, 1),
i(5, halt),
c(1).
```

Figure 6.6: Assembler code and corresponding RAM machine query

Complexity-wise completeness in practice

In principle, every algorithm can be implemented in CHR using the RAM simulator program. Of course, this does not result in a natural and elegant CHR program, but at least the resulting CHR program has the right time and space complexity.

Consider the following very simple C program:

```
long a=1, b=1000000, c=0;
while(b != 0) {
   c += a;
   b -= a;
}
```

This C program corresponds to the Intel assembler code shown in Figure 6.6 (on the left hand side). The assembler code can also be seen as a query for the RAM machine program RAMSIMUL, as shown in the right hand side of Figure 6.6. The observant reader will notice that this is in fact just the "Loop" benchmark that was discussed in the previous section.

By translating assembler code to a RAM simulator query, we get a CHR program with the same asymptotic time and space complexity: in this example, both the CHR program and the assembler code take linear time and constant space.

Although the CHR(Prolog) RAM simulator executes such RAM programs with the correct asymptotic complexity, the execution time is about ten thousand times larger than that of the original assembler code program for this example: the RAM simulator takes about 10 seconds while the assembler program runs in 1.6 milliseconds. In other words, the computational power of a Pentium 4 is reduced to that of a Commodore 64.

Of course no sane programmer would write CHR programs in this way — not just because of the debilitating slowdown: such programs also lack desirable properties of CHR programs (conciseness, readability, adaptability, incrementality, concurrency, ...) that are often obtained naturally in hand-written CHR programs. Hence, it remains necessary to manually construct CHR programs.

Experimental evaluation

Chapter 5 of [Sneyers, 2008b] discussed several CHR programs that implement classical algorithms. In this section we investigate the performance of two of these hand-written CHR programs. We compare their performance to that of an efficient reference implementation, in the low-level language C, of the same algorithms. The goal is to obtain an estimate of the (constant factor) performance penalty for using CHR.

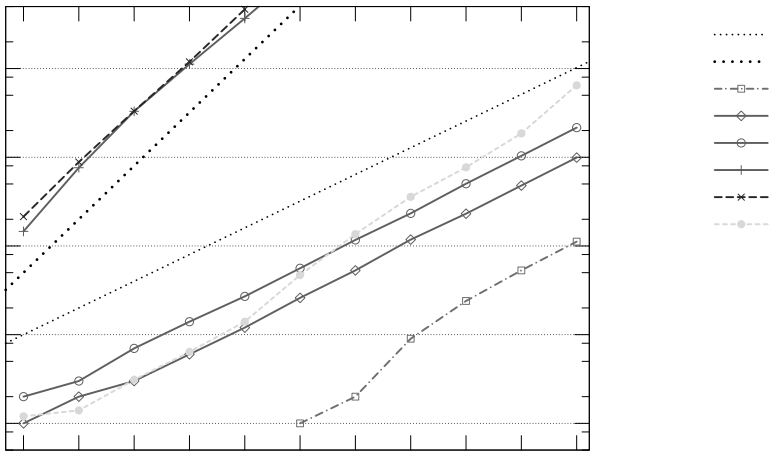


Figure 6.7: Results of the "Union-find" benchmark

As we have seen in Chapter 3, the Leuven CHR system allows the programmer to specify optional type and mode declarations for the constraint arguments (see also Table 7.1 in [Sneyers, 2008b]). The information is used for optimizing compilation. The original CHR system in SICStus Prolog, by Holzbaur and Frühwirth [1998], does not have a mechanism to provide such declarations. In the Java CHR system, precise type declarations are obligatory since Java is a typed language.

Union-find. Consider again the union-find program of Schrijvers and Frühwirth [2006]. Table B.6 in [Sneyers, 2008b] lists the execution times for this CHR program in the different CHR systems. We compare these results against a very efficient C implementation[7]. In order to achieve the optimal complexity, type detail level 1 is needed (see Table 7.1 in [Sneyers, 2008b]). With type detail level 2, the high-level CHR(hProlog) implementation is roughly 10 times slower than the direct low-level implementation in C. The results are plotted in Figure 6.7. The benchmark consists of performing n `make` operations, followed by n random `union` operations and n random `find` operations.

To get an idea of the constant factors involved in space usage, consider the following numbers. The C program uses only one array to represent the entire data structure; every element takes one word (4 bytes): positive integers represent the index of the parent, negative integers represent the rank of a root. In contrast, the CHR(hProlog) program uses 9 words to represent an element: two arrays are used (one for roots, one for non-roots), which contain pointers to 7-word suspension terms: one word for the wrapper functor, two for the constraint arguments, one for the identifier, three for the state (see also [Sneyers, 2008b], chapter 4.2.1). Hence, for the union-find algorithm, the CHR version uses about ten times as much space as the C version.

Dijkstra's algorithm with Fibonacci heaps. In [Sneyers, 2008b], chapter 5.2, we have discussed a CHR implementation of the shortest path algorithm of Dijkstra [1959],

[7] Written by Ariel Faigon, based on a version by Robert Sedgewick. The source code is available at `http://www.yendor.com/programming/minauto/ufind.c`

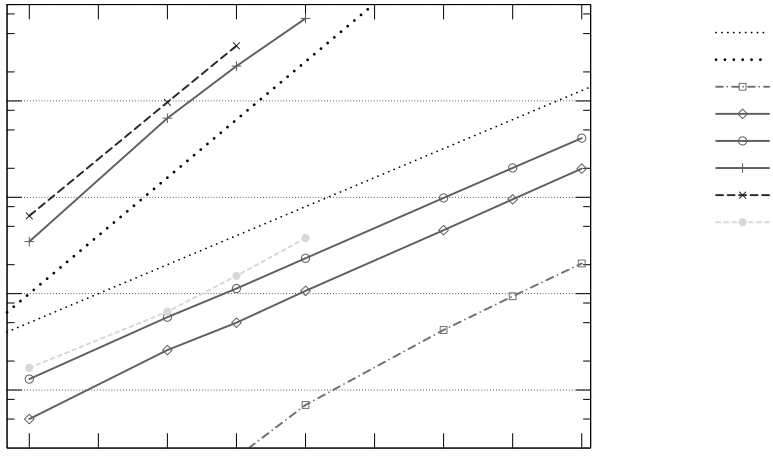


Figure 6.8: Results of the "Dijkstra" benchmark

which uses the Fibonnacci heaps data structure of Fredman and Tarjan [1987]. We have compared the performance of the DIJKSTRA program to that of a C implementation[8] by Cherkassky et al. [1996]. The results are listed in Table B.7 in [Sneyers, 2008b] and they are plotted in Figure 6.8.

Note the overall resemblance between Figure 6.8 and Figure 6.7. Again, type detail level 1 (see Table 7.1 in [Sneyers, 2008b]) is needed to achieve optimal complexity. Without any information, the program exhibits a quadratic time complexity, because the default data structure does not allow constant time look-ups of ground constraints. When groundness information is available, the optimal $O(n \log n)$ time complexity is achieved. The time gap between the hProlog CHR program (type detail level 2) and the C program is — again — a constant factor of about 10.

The difference in space usage is less pronounced than for the union-find algorithm. The CHR program uses about three times more space than the C program: for an input size of 256k nodes, the C program uses 23 megabytes, while the CHR(hProlog) program needs 63 megabytes.

Conclusion: CHR(hProlog) / C ≈ 10

The above results indicate that the constant time factor separating CHR(hProlog) from C is approximately 10. In terms of space usage, the constraint representation has a fixed overhead: in programs using very light-weight data representations (e.g. union-find), this results in a relatively large constant space factor (e.g. 10); in programs with more complicated representations (e.g. Fibonacci heaps), the constant space factor is smaller (e.g. 3). Future and ongoing work in CHR compiler optimization will further reduce these factors. In particular, the space overhead can be much reduced by further specializing the constraint representation.

Extrapolating from the above examples, combined with the results of the previous section, we can summarize our results as follows: *"The current state of the art in CHR and Prolog systems suffices to implement any algorithm in CHR(Prolog), in a natural and*

[8] The source code is available at: `http://www.avglab.com/andrew/soft.html`

high-level way, with a time and space complexity which is within a constant factor of 10 from the best-known implementation in any other programming language." Clearly, the notion of "natural and high-level" is rather vague and open for interpretation, so unlike the complexity-wise completeness result of Corollary 6.3.9, the above statement cannot be proved mathematically.

Future work. More empirical evidence can be gathered by implementing more algorithms in CHR and comparing the performance with that of implementations in low-level programming languages. One could for instance attempt to implement, in CHR, a (large) subset of the algorithms described in classic books on algorithms, for example [Knuth, 1997-1998]. Perhaps this approach allows to make conclusions about the kind of algorithms for which CHR is more or less suitable.

A more direct approach would be to construct a direct translation of some (subset) of an imperative language (e.g. some subset of Java) to CHR, and measure the constant factor gap — knowing that this gap will be larger for a general translation scheme than for hand-tailored 'free translations'.

Bibliography

George M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.

Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman, 1975.

Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, et al. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, 1984. ISBN 0-387-15011-0.

Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(4):269–271, 1959.

Gregory J. Duck and Tom Schrijvers. Accurate functional dependency analysis for Constraint Handling Rules. In T. Schrijvers and Th. Frühwirth, editors, *CHR'05*, K.U.Leuven, Dept. Comp. Sc., Technical report CW421, pages 109–124, Sitges, Spain, 2005. URL `http://www.cs.kuleuven.be/~dtai/projects/CHR/biblio/chr2005/duck_schr_accurate_funcdep_chr05.ps`.

Steven Eker. Associative-commutative rewriting on large terms. In Robert Nieuwenhuis, editor, *RTA'03: Rewriting Techniques and Applications*, volume 2706 of *LNCS*, pages 14–29, Valencia, Spain, June 2003. Springer.

Charles L. Forgy. Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

Michael Fredman and Robert Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

Ernest Friedman-Hill. *Jess in Action: Java Rule-Based Systems*. Manning, 2003.

Thom Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In A. Di Pierro and H. Wiklicky, editors, *QAPL'01: Proc. First Intl. Workshop on Quantitative Aspects of Programming Languages*, volume 59(3) of *ENTCS*, Florence, Italy, 2002a.

Thom Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *KR'02: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning*, pages 547–557, Toulouse, France, April 2002b. Morgan Kaufmann.

Maurizio Gabbrielli, Jacopo Mauro, Maria Chiara Meo, and Jon Sneyers. Decidability properties for fragments of chr. *TPLP*, 10(4-6):611–626, 2010.

Joseph C. Giarratano and Gary Riley. *Expert Systems: Principles and Programming*. PWS Publishing Co., 1994. ISBN 0534937446.

Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Trans. American Mathematical Society*, 117:285–306, May 1965.

Christian Holzbaur and Thom Frühwirth. Constraint Handling Rules reference manual, release 2.2. Technical Report TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria, 1998.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman, 2001.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *HOPL-III: Proc. 3rd ACM SIGPLAN Conf. History of Programming Languages*, pages 1–55, San Diego, CA, USA, June 2007.

Donald E. Knuth. *The Art of Computer Programming, Volumes 1–3*. Addison-Wesley, 1997-1998.

Nancy Mazur. *Compile-time Garbage Collection for the Declarative Language Mercury*. PhD thesis, K.U.Leuven, Belgium, May 2004.

Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.

Donald R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321479.321481.

Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, Baltimore, MD, USA, September 1998.

John E. Savage. *Models of Computation*. Addison-Wesley Longman, 1998.

Walter J. Savitch. The influence of the machine model on computational complexity. In J.K. Lenstra, A.H.G. Rinnooy Kan, and P. van Emde Boas, editors, *Interfaces between Computer Science and Operations Research*, volume 99 of *Mathematical Centre Tracts*, pages 1–32. 1978.

Tom Schrijvers and Thom Frühwirth. Optimal union-find in Constraint Handling Rules. *TPLP*, 6(1–2):213–224, 2006. ISSN 1471-0684. doi: http://dx.doi.org/10.1017/S1471068405002541.

Jon Sneyers. Turing-complete subclasses of CHR. In María García de la Banda and Enrico Pontelli, editors, *ICLP'08*, LNCS, pages 759–763, Udine, Italy, December 2008a. Springer.

Jon Sneyers. *Optimizing Compilation and Computational Complexity of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, November 2008b.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. Memory reuse for CHR. In S. Etalle and M. Truszczynski, editors, *ICLP'06*, volume 4079 of *LNCS*, pages 72–86, Seattle, Washington, August 2006. Springer.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming*, 29(1-3):17–64, 1996.

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

Peter van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier, 1990.

Chapter 7

Complexity Analysis of $\mathrm{CHR}^{\mathrm{rp}}$ Programs

Author:	Leslie De Koninck
Thesis Title:	Execution Control for Constraint Handling Rules
School:	K.U.Leuven, Belgium
Publication Year:	2008

Foreword

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

This chapter is part of a line of research on deriving theoretical properties of CHR programs. It studies the derivation of the time complexity of CHR programs and has a similar aim as [Frühwirth, 2002a,b]. Sneyers et al. [2008] derives the time and space complexity of a RAM machine simulator in CHR and as such shows that any algorithm can be implemented in CHR with the best known time and space complexity. The derivation of other theoretical properties of CHR programs have been studied in other work, most notably confluence [Abdennadher, 1997, Duck et al., 2007] and termination [Frühwirth, 2000, Pilozzi et al., 2007, Voets et al., 2007]. Also related is work on theoretical properties of CHR as a language. We already mentioned [Sneyers et al., 2008] but also refer to [Sneyers, 2008, Di Giusto et al., 2009, Gabbrielli et al., 2010, 2009] which discuss the expressiveness of various subsets and extensions of CHR.

7.1 Introduction

Sneyers et al. [2008] have shown that any algorithm can be implemented in CHR while preserving both time and space complexity.

In "Logical Algorithms" (LA) [Ganzinger and McAllester, 2002] (and based on previous work in [Ganzinger and McAllester, 2001, McAllester, 1999]), a bottom-up logic programming language is presented for the purpose of facilitating the derivation of complexity results of algorithms described by logical inference rules. This problem is far from trivial because the runtime is not necessarily proportional to the derivation length (i.e., the number of rule applications), but also includes the cost of pattern matching for multi-headed rules, as well as costs related to high-level execution control which is specified using rule priorities in the Logical Algorithms language. The language of Ganzinger and McAllester resembles CHR in many ways and has often been referred to in the discussion of complexity results of CHR programs [Christiansen, 2005, Frühwirth, 2002b, Schrijvers and Frühwirth, 2006, Sneyers et al., 2006a]. In particular, Christiansen [2005] uses the meta-complexity theorem that accompanies the Logical Algorithms language, and notes that the CHR system used (SICStus CHR by Holzbaur and Frühwirth [1998]) does not always exhibit the right complexity because previously computed partial rule matches are not stored.

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

Our current implementation of CHR$^{\text{rp}}$ as presented in Chapter 4 does not offer the complexity guarantees required for the meta-complexity theorem for Logical Algorithms to hold via translation to CHR$^{\text{rp}}$. Another issue is that the translation from CHR$^{\text{rp}}$ to Logical Algorithms is restricted to a subset of CHR$^{\text{rp}}$. Therefore, we propose a new implementation of CHR$^{\text{rp}}$, designed such that it supports a new meta-complexity theorem for the complete CHR$^{\text{rp}}$ language, while also ensuring that LA programs translated into CHR$^{\text{rp}}$ are executed with the required complexity. We note that this alternative implementation is not optimized for average case performance, but is designed to achieve certain complexity guarantees.

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

Overview The rest of this chapter is organized as follows. In Section 7.2, the syntax and semantics of the Logical Algorithms language is reviewed and the known meta-complexity theorems for both LA and CHR are presented. In Section 7.3 a translation of LA programs to CHR$^{\text{rp}}$ programs is presented and in Section 7.4, the opposite is done for a subset of CHR$^{\text{rp}}$. Section 7.5 proposes an alternative implementation for CHR$^{\text{rp}}$ which enables a

new meta-complexity theorem for this language, given in Section 7.6. Some concluding remarks are given in Section 7.7.

7.2 Logical Algorithms and CHR$^{\mathrm{rp}}$

In this section, we give an overview of the syntax and semantics of Logical Algorithms (Section 7.2.1) and review the meta-complexity results that are known for both LA and CHR (Section 7.2.2).

7.2.1 Logical Algorithms

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

Syntax

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

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

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

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

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

Example 7.2.1. *An example rule (from Dijkstra's shortest path algorithm as presented in [Ganzinger and McAllester, 2002]) with name* `d2` *and priority 1 is*

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

The antecedent `D2 < D1` *is a comparison, the atoms* $\texttt{dist}(\texttt{V}, \texttt{D}_1)$ *and* $\texttt{dist}(\texttt{V}, \texttt{D}_2)$ *are positive user-defined antecedents. The negative ground atom* $\texttt{del}(\texttt{dist}(\texttt{a}, 5))$ *is an example of a negative assertion.*

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

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

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

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

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

Table 7.1: The Logical Algorithms operational semantics

Operational Semantics

A Logical Algorithms execution state σ consists of a set of (positive and negative) assertions. A state can simultaneously contain the positive assertion A and the negative assertion $\texttt{del}(A)$. Let $\mathcal{D}$ be the usual interpretation for the comparisons. Table 7.1 shows the (single) transition of the Logical Algorithms operational semantics for a given program P.

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

Differences compared to CHR^{rp}

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

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

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

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

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

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

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

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

7.2.2 Meta-Complexity Results for LA and CHR$^{\text{rp}}$

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

The Logical Algorithms Meta-Complexity Result

A *prefix instance* of a Logical Algorithms rule $r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$ is a tuple $\langle \theta(r), i \rangle$ with θ a ground substitution defined on the variables occurring in $A_1, \ldots, A_i$ and $1 \leq i \leq n$. Its antecedents are $\theta(A_1), \ldots, \theta(A_i)$. A *strong prefix firing* is a prefix instance whose antecedents hold in a state with priority lower or equal to the prefix' rule priority.[2] The time complexity for running Logical Algorithms programs is given by Ganzinger and McAllester [2002] as $\mathcal{O}(|\sigma_0| + P_{\mathsf{s}} + (P_{\mathsf{d}} + A_{\mathsf{d}}) \cdot \log N)$ where σ_0 is the initial state and $|\sigma_0|$ is its size. P_{s} is the number of strong prefix firings of static priority rules and P_{d} is the number of strong prefix firings of dynamic priority rules; A_{d} is the number of assertions that may participate in a dynamic priority rule instance; and N is the number of distinct priorities. The following example is adapted from [Ganzinger and McAllester, 2002].

Example 7.2.2 (Dijkstra's Shortest Path)**.** *Listing 7.1 shows an implementation of Dijkstra's single-source shortest path algorithm in LA.*

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

Listing 7.1: Dijkstra's shortest path algorithm in Logical Algorithms

The code is very similar to the CHR$^{\text{rp}}$ *code of Listing 4.2. A* `source`(v) *fact means that* v *is the (unique) source node for the algorithm. A* `dist`(v, d) *fact means that the shortest path distance from the source node to node* v *does not exceed* d*. Finally, an* `e`(v, c, u) *fact means there is an edge from node* v *to node* u *with cost (weight)* c*. Given an initial state consisting of one* `source`*/1 fact and* e `e`*/3 facts, we can derive that the number of strong prefix firings is* $\mathcal{O}(1)$ *for rule* `d1`*, and* $\mathcal{O}(e)$ *for both rules* `d2` *and* `d3`*. This result is based on the fact that at priority 2 and lower, there is at most one (positive)* `dist`*/2 fact for each node, and each of these facts represent the shortest path distance from the source node to this node. This means that at most* e `dist`*/2 facts are ever created. Using the meta-complexity theorem, we find that the total complexity is* $\mathcal{O}(e \log e)$*.*

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

The "As Time Goes By" Approach

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

$$\mathcal{O}\left(D\sum_{r\in P}(c_{\mathsf{max}}^{n_r}(O_{H_r}+O_{G_r})+(O_{C_r}+O_{B_r}))\right) \tag{7.1}$$

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

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

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

Example 7.2.3 (Boolean)**.** *The rules below implement the Boolean* $\texttt{and}(x, y, x \wedge y)$ *constraint given that 1 represents* true *and 0 represents* false*.*

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

Let the rank of an and*/3 constraint be one, then the rank of the head of each rule equals one, and the rank of the body equals zero.*[4] *For a goal consisting of* n and*/3 constraints, the derivation length is* n*, which is also the maximal number of CHR constraints in the store. The cost of head matching, (implicit) guard checking, removing CHR constraints and asserting built-in constraints can all be considered constant. Then using* (7.1)*, we derive that the total runtime complexity is* $\mathcal{O}(n^2)$*.*

A First Comparison

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

[3] Built-in constraints may lead to a worse complexity in practical optimized implementations if many constraints are repeatedly reactivated without this resulting in new rule applications. We return to this issue in Section 7.6.3.

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

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

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

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

$$\mathcal{O}\left(D \cdot \sum_{r \in P} c_{\mathsf{max}}^{n_r}\right) \tag{7.2}$$

In the absence of (dynamic) priorities, the total runtime complexity according to the Logical Algorithms meta-complexity result is bounded by the same formula (7.2) and hence is at least as accurate as the result of Frühwirth [2002b] given that the cost of both head matching (O_{H_r}) and adding and removing CHR constraints (O_{B_r}) is constant for each rule r.

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

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

7.3.1 Translation Schema

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

Set and Deletion Semantics

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

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

If a representation already exists, one of the priority 1 rules updates this representation. Otherwise, one of the priority 2 rules generates a new representation. At lower priorities, it is guaranteed that every assertion, whether asserted positively, negatively or both, is represented by exactly one constraint in the store.

Rules

Given a LA rule $r \in P$ of the form

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

We first split up the antecedents into user-defined antecedents and comparison antecedents by using the split function defined below.

$$\begin{aligned}\mathsf{split}([A|T]) = &\begin{cases}\langle [A|A^u], A^c\rangle & \text{if } A \text{ is a user-defined atom}\\ \langle A^u, [A|A^c]\rangle & \text{if } A \text{ is a comparison}\end{cases}\\ &\text{where } \mathsf{split}(T) = \langle A^u, A^c\rangle\\ \mathsf{split}(\epsilon) = &\langle \epsilon, \epsilon\rangle\end{aligned}$$

In the Logical Algorithms language, a given assertion may participate multiple times in the same rule instance, whereas in CHR all constraints in a single rule instance must be different. To overcome this semantic difference, a single LA rule is translated as a set of CHR rules such that every CHR rule covers a case of syntactically equal head constraints. Let $\langle A^u, A^c\rangle = \mathsf{split}([A_1, \ldots, A_n])$ with $A^u = [A^u_1, \ldots, A^u_m]$ and $A^c = [A^c_1, \ldots, A^c_l]$. Let $\mathcal{P}$ be the set of all partitions of $\{1, \ldots, m\}$.[5] For a given partition $\rho \in \mathcal{P}$, the following function returns the most general unifier that unifies all antecedents $\{A_i \mid i \in S\}$ for every $S \in \rho$ where $\mathsf{mgu}(S)$ is the most general unifier of all elements in S.

$$\mathsf{partition\_to\_mgu}(\rho, [A^u_1, \ldots, A^u_m]) = \mathop{\circ}_{S\in\rho} \mathsf{mgu}(\{A^u_i \mid i \in S\})$$

Let $\mathcal{PU} = \{\langle \rho, \theta\rangle \mid \rho \in \mathcal{P} \wedge \theta = \mathsf{partition\_to\_mgu}(\rho, A^u) \wedge \mathcal{D} \models \bar{\exists}_\emptyset \theta(A^c)\}$. $\mathcal{PU}$ contains all partitions for which partition_to_mgu is defined and for which the comparison antecedents A^c are still satisfiable after applying the unifier. The next step is to filter out antecedents so that every set in the partition has only one representative. This is done by computing $\mathsf{filter}(\theta(A^u), \rho)$ for each $\langle \rho, \theta\rangle \in \mathcal{PU}$ where the filter function is as follows:

$$\begin{aligned}\mathsf{filter}([\theta(A^u_i)|T], \rho) = &\begin{cases}[\theta(A^u_i)|\mathsf{filter}(T, \rho)] & \text{if } \exists S \in \rho : i = \min(S)\\ \mathsf{filter}(T, \rho) & \text{otherwise}\end{cases}\\ \mathsf{filter}(\epsilon, \_) = &\epsilon\end{aligned}$$

Finally, we add mode indicators to all remaining user-defined antecedents:

$$\begin{aligned}\mathsf{modes}([A^{u'}|T]) = &\begin{cases}\langle [a_{\mathtt{r}}(\bar{X}, \mathtt{p})|A^m], N\rangle & \text{if } A^{u'} = a(\bar{X})\\ \langle [a_{\mathtt{r}}(\bar{X}, N')|A^m], [N' \neq \mathtt{p}|N]\rangle & \text{if } A^{u'} = \mathtt{del}(a(\bar{X}))\end{cases}\\ &\text{where } \langle A^m, N\rangle = \mathsf{modes}(T)\\ \mathsf{modes}(\epsilon) = &\langle \epsilon, \epsilon\rangle\end{aligned}$$

The modes function returns both the resulting antecedents and the necessary conditions on the mode indicators of these antecedents. For every $\langle \rho, \theta\rangle \in \mathcal{PU}$, the CHR translation $T_R(P)$ contains a rule

$$p + 2 :: r_\rho \ @ \ H \implies g_1, g_2 \mid C'$$

where $\langle H, g_1\rangle = \mathsf{modes}(\mathsf{filter}(\theta(A^u), \rho))$, $g_2 = \theta(A^c)$ and $C' = \theta(C)$.

[5] $\mathcal{P}$ contains B_m elements in the worst case with B_m the m^{th} Bell number.

Examples

We illustrate the translation schema on some examples.

Example 7.3.1. *The translation of the LA implementation of Dijkstra's shortest path algorithm given in Listing 7.1 is given in Listing 7.2.*

```
1 :: e_r(V,C,U,M) \ e(V,C,U) <=> M \= n | true.
1 :: e_r(V,C,U,n) , e(V,C,U) <=> e_r(V,C,U,b).
2 ::                e(V,C,U) <=> e_r(V,C,U,p).

1 :: e_r(V,C,U,M) \ del(e(V,C,U)) <=> M \= p | true.
1 :: e_r(V,C,U,p) , del(e(V,C,U)) <=> e_r(V,C,U,b).
2 ::                del(e(V,C,U)) <=> e_r(V,C,U,n).

... % (similar rules for source/1 and dist/2)

  3 :: d1_1   @ source_r(V,p) ==> dist(V,0).
  3 :: d2_1/2 @ dist_r(V,D_1,p), dist_r(V,D_2,p) ==> D_2 < D_1 | del(dist(V,D_1)).
D+4 :: d3_1/2 @ dist_r(V,D,p), e_r(V,C,U,p) ==> dist(U,D+C).
```

Listing 7.2: Translation to CHR^{rp} of the program of Listing 7.1

Example 7.3.2. *The following rule is part of the union-find implementation given in [Ganzinger and McAllester, 2002].*

```
uf4 @ 1 : union(X,Y), find(X,Z), find(Y,Z) => del(union(X,Y)).
```

Because antecedents $\texttt{find}(\texttt{X},\texttt{Z})$ *and* $\texttt{find}(\texttt{Y},\texttt{Z})$ *are unifiable, their translation to* CHR^{rp} *is as follows:*

```
3 :: uf4_1/2/3 @ union_r(X,Y,p), find_r(X,Z,p), find_r(Y,Z,p) ==> del(union(X,Y)).
3 :: uf4_1/23  @ union_r(X,X,p), find_r(X,Z,p) ==> del(union(X,X)).
```

7.3.2 Correspondence between LA and ω_p Derivations

In this subsection, we show that every derivation of the original program under the Logical Algorithms semantics, corresponds to a derivation of the translation under the ω_p semantics of CHR^{rp}. In order to do so, we introduce a mapping function $\mathsf{chr\_to\_la}$ between *reachable* CHR execution states and Logical Algorithms states.[6] Reachability is considered with respect to initial states of the form $\langle G, \emptyset, \texttt{true}, \emptyset\rangle_n$ where the user-defined constraints in G are of the form $a(\bar{X})$ and $\texttt{del}(a(\bar{X}))$ and do not include constraints of the form $a_{\texttt{r}}(\bar{X}, M)$.

$$\begin{aligned}\mathsf{chr\_to\_la}(\sigma) = \{&a(\bar{X}) \mid a(\bar{X}) \in A \vee (a_{\texttt{r}}(\bar{X}, M) \in A \wedge M \neq \texttt{n})\}\\ \cup\ \{&\texttt{del}(a(\bar{X})) \mid \texttt{del}(a(\bar{X})) \in A \vee (a_{\texttt{r}}(\bar{X}, M) \in A \wedge M \neq \texttt{p})\}\end{aligned}$$

where $\sigma = \langle G, S, B, T\rangle_n$ and $A = G \cup \mathsf{chr}(S)$. The mapping function also takes into account the constraints that are still in the goal and those for which the set and deletion semantics rules have not yet fired. In the rest of this section, we first show how CHR execution states are normalized and then show that in a Logical Algorithms state and its corresponding normalized CHR execution state, corresponding rule instances can fire. We start by defining a *pre-normal form.*

[6] See [Duck et al., 2007] for a formal definition of reachability.

Definition 7.3.3 (Pre-normal Form)**.** *A (reachable) state σ is in pre-normal form if and only if $\sigma = \langle \emptyset, S, \texttt{true}, T\rangle_n$, all constraints in S are of the form $a_r(\bar{X}, M)\#i$, and if $a_r(\bar{X}, M_1)\#i_1 \in S$ and $a_r(\bar{X}, M_2)\#i_2 \in S$ then $i_1 = i_2$ (and consequently $M_1 = M_2$).*

The following lemma shows that every reachable state is *pre-normalized* before rules are tried with priority > 2.

Lemma 7.3.4 (Pre-normalization)**.** *For every reachable state σ, there exists a finite derivation $D = \sigma \overset{\omega_p}{\rightarrowtail}^*_{T(P)} \sigma^*$ such that σ^* is in pre-normal form, $\mathsf{chr\_to\_la}(\sigma)$=$\mathsf{chr\_to\_la}(\sigma^*)$, and all rules fired in D have priority 1 or 2. Every state has a unique pre-normal form with respect to the $\mathsf{chr\_to\_la}$ mapping function.*

Proof. We introduce the following ranking function on CHR states:

$$\|\sigma\| = 2 \cdot \left|\{a(\bar{X}) \mid a(\bar{X}) \in A\} \uplus \{\texttt{del}(a(\bar{X})) \mid \texttt{del}(a(\bar{X})) \in A\}\right| + |G|$$

where $\sigma = \langle G, S, \texttt{true}, T\rangle_n$, $A = G \uplus \mathsf{chr}(S)$ and if X is a (multi-)set, $|X|$ is its cardinality. Clearly, the rank of any state is positive, and if $\|\sigma\| = 0$, state σ is in pre-normal form. If σ is not in pre-normal form, then there exists at least one transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$. We show that for all such transitions $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ and $\|\sigma'\| < \|\sigma\|$, which ensures termination.

If the goal G is not empty, then only the **Introduce** transition is applicable. Every application of this transition moves a CHR constraint from the goal to the CHR constraint store, so $\|\sigma'\| = \|\sigma\| - 1$. By definition, $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$ because the $\mathsf{chr\_to\_la}$ function does not distinguish between the goal and the CHR constraint store.

If the goal G is empty then given that σ is not in pre-normal form, $\mathsf{chr}(S)$ contains a constraint of the form $a(\bar{X})$ or $\texttt{del}(a(\bar{X}))$. We look into detail to the case of $a(\bar{X}) \in \mathsf{chr}(S)$; the case of $\texttt{del}(a(\bar{X})) \in \mathsf{chr}(S)$ is similar. We start by showing that at least one rule of priority 1 or 2 is applicable. Next, we show that each rule application decreases the norm and maintains the invariance with respect to the $\mathsf{chr\_to\_la}$ function.

Assume $a(\bar{X}) \in \mathsf{chr}(S)$. If $a_{\mathtt{r}}(\bar{X}, \mathtt{p}) \in \mathsf{chr}(S)$ or $a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \in \mathsf{chr}(S)$ then the following rule of $T(P)$ is applicable:

$$1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus a(\bar{X}) \iff M \neq \mathtt{n} \mid \texttt{true}$$

If $a_{\mathtt{r}}(\bar{X}, \mathtt{n}) \in \mathsf{chr}(S)$ then the rule below applies:

$$1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{n}), a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{b})$$

Finally, if no rule of priority 1 can be applied, which implies that no constraint of the form $a_{\mathtt{r}}(\bar{X}, M) \in \mathsf{chr}(S)$, then the following $T(P)$ rule can fire:

$$2 :: a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{p})$$

This covers all possibilities. Now we look at what happens after firing one of the priority 1 or 2 rules. The rule

$$1 :: a_{\mathtt{r}}(\bar{X}, M) \setminus a(\bar{X}) \iff M \neq \mathtt{n} \mid \texttt{true}$$

removes a constraint $a(\bar{X})\#i$ from S and has an empty body, so $\|\sigma'\| = \|\sigma\| - 2$. Since $M \neq \mathtt{n}$ the removed constraint was already represented by the $a_{\mathtt{r}}(\bar{X}, M)$ constraint and so $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$. Firing

$$1 :: a_{\mathtt{r}}(\bar{X}, \mathtt{n}), a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{b})$$

causes the removal of two constraints from S, namely $a_{\mathtt{r}}(\bar{X}, \mathtt{n})\#i$ and $a(\bar{X})\#j$. Furthermore, it adds a new constraint $a_{\mathtt{r}}(\bar{X}, \mathtt{b})$ to G. This results in $\|\sigma'\| = \|\sigma\| - 1$. The new constraint represents the combined mode of both removed constraints and hence $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$. Finally, the rule

$$2 :: a(\bar{X}) \iff a_{\mathtt{r}}(\bar{X}, \mathtt{p})$$

is only applicable if $\mathsf{chr}(S)$ does not contain a constraint of the form $a_{\mathtt{r}}(\bar{X}, M)$. It removes a constraint $a(\bar{X})\#i$ from S and adds a new constraint $a_{\mathtt{r}}(\bar{X}, \mathtt{p})$ to G, resulting in $\|\sigma'\| = \|\sigma\| - 1$. The new representation covers the positive assertion and so $\mathsf{chr\_to\_la}(\sigma') = \mathsf{chr\_to\_la}(\sigma)$.

In summary, if the goal is empty and σ is not in pre-normal form, a rule of priority 1 or 2 can fire and so no rule with lower priority is applicable. All applicable transitions strictly decrease the value of the ranking function and so the pre-normalization terminates. Finally, none of the possible transitions changes the value of $\mathsf{chr\_to\_la}$. □

The state σ^* is called a pre-normalization of σ.

Definition 7.3.5 (Implied Rule Instance)**.** *A rule instance $\theta(r)$ is implied in a state σ if $\theta(C) \subseteq \mathsf{chr\_to\_la}(\sigma)$ with $\theta(C)$ the conclusion of $\theta(r)$.*

Lemma 7.3.6 (Normalization)**.** *Let there be given a pre-normalized state $\sigma = \langle \emptyset, S, \mathtt{true}, T\rangle_n$. If there exists a transition $\sigma \overset{\omega_p}{\mapsto}_{T(P)} \sigma'$ in which an implied rule instance fires, then the pre-normalization of σ' has the form $\langle \emptyset, S, \mathtt{true}, T'\rangle_{n'}$ with $T' \supsetneq T$. In other words $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ and the CHR constraint store after pre-normalization is unchanged from the one before the implied rule instance fired while the propagation history is increased.*

Proof. Let $\theta(r)$ be the implied rule instance with conclusion $\theta(C)$. As $\theta(C) \subseteq \mathsf{chr\_to\_la}(\sigma)$ with $\sigma = \langle \emptyset, S, \mathtt{true}, T\rangle_n$, we have $\sigma' = \langle \theta(C), S, \mathtt{true}, T \cup \{t\}\rangle_n$ and $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ with t the propagation history tuple corresponding to $\theta(r)$. The goal G of σ' equals $\theta(C)$ and so it holds that if $a(\bar{X}) \in G$ then $a_{\mathtt{r}}(\bar{X}, \mathtt{p}) \in \mathsf{chr}(S)$ or $a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \in \mathsf{chr}(S)$ and if $\mathtt{del}(a(\bar{X})) \in G$ then $a_{\mathtt{r}}(\bar{X}, \mathtt{n}) \in \mathsf{chr}(S)$ or $a_{\mathtt{r}}(\bar{X}, \mathtt{b}) \in \mathsf{chr}(S)$. Now all constraints in the goal are first introduced in the CHR constraint store. Next, the newly introduced CHR constraints are removed one by one using one of the following rules:

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

These rules remove all the constraints that were introduced from the goal and do not change the rest of the CHR constraint store, hence after pre-normalization, the CHR constraint store equals that of state σ again. □

Because the CHR constraint store remains unchanged after firing an implied rule instance and pre-normalizing the resulting state, only finitely many such rule instances can fire before either reaching a final execution state, or a state in which a non-implied rule instance can fire. We call such a state *normalized*.

Definition 7.3.7 (Normal Form)**.** *A pre-normalized CHR execution state σ is in normal form if it is a final state ($\sigma \overset{\omega_p}{\not\mapsto}_{T(P)}$) or there exists a transition $\sigma \overset{\omega_p}{\mapsto}_{T(P)} \sigma'$ such that $\mathsf{chr\_to\_la}(\sigma') \not\supseteq \mathsf{chr\_to\_la}(\sigma)$, i.e., in which a non-implied rule instance is fired.*

Lemma 7.3.8. *For every Logical Algorithms state σ_{LA} and every normalized CHR execution state $\sigma = \langle \emptyset, S, \texttt{true}, T \rangle_n$ such that $\sigma_{LA} = \mathsf{chr\_to\_la}(\sigma)$, there exists a transition $\sigma_{LA} \overset{LA}{\mapsto}_P \sigma'_{LA}$ if and only if there exists a transition $\sigma \overset{\omega_p}{\mapsto}_{T(P)} \sigma'$ firing a non-implied rule instance such that $\sigma'_{LA} = \mathsf{chr\_to\_la}(\sigma')$.*

Proof. A transition of σ_{LA} to σ'_{LA} implies there exists an applicable rule instance $\theta(r)$ of a rule r in P with priority p of the form

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

Let $\langle A^u, A^c \rangle = \langle [A^u_1, \ldots, A^u_m], [A^c_1, \ldots, A^c_l] \rangle = \mathsf{split}([A_1, \ldots, A_n])$ where we use the split function defined in Section 7.3.1. The user-defined antecedents can be partitioned into sets of syntactically equal antecedents with respect to the matching substitution θ. The following function returns this partition:

$$\mathsf{substitution\_to\_partition}(\theta, [A^u_1, \ldots, A^u_m]) = \{S_1, \ldots, S_m\}$$

where $S_i = \{j \mid \theta(A^u_i) = \theta(A^u_j)\}$. Let $\rho = \mathsf{substitution\_to\_partition}(\theta, A^u)$. From the partition, we find the most general unifier θ' that unifies all antecedents $\{A^u_i \mid i \in S\}$ for every $S \in \rho$: $\theta' = \mathsf{partition\_to\_mgu}(\rho, A^u)$ with $\mathsf{partition\_to\_mgu}$ as defined in Section 7.3.1. Clearly, θ' exists and is more general than θ. The applicability of the **Apply** transition means that for all comparison antecedents A^c_i with $1 \leq i \leq l$, $\mathcal{D} \models \theta(A^c_i)$ and so it holds that $\mathcal{D} \models \bar{\exists}_\emptyset \theta'(A^c_1 \wedge \ldots \wedge A^c_l)$ and consequently a rule r_ρ exists. This rule looks as follows:

$$p + 2 :: r_\rho \ @ \ H_1, \ldots, H_k \implies g_1, g_2 \mid C'$$

with $\langle [H_1, \ldots, H_k], g_1 \rangle = \mathsf{modes}(A^f)$, $A^f = [A^f_1, \ldots, A^f_k] = \mathsf{filter}(\theta'(A^u), \rho)$, $g_2 = \theta'(A^c)$ and $C' = \theta'(C)$. The modes and filter functions are as defined in Section 7.3.1.

Let θ'' be a ground matching substitution such that $\theta = \theta''|_{\mathsf{vars}(\theta)} \circ \theta'$ where $\theta''|_{\mathsf{vars}(\theta)}$ is the projection of θ'' on the variables in θ. Since θ' is more general than θ, θ'' exists. For all $i \in \{1, \ldots, k\}$, if $A^f_i = a(\bar{X})$ then $H_i = a_{\mathtt{r}}(\bar{X}, \mathtt{p})$. Because of the applicability of Logical Algorithms rule r in state σ_{LA}, $\theta''(a(\bar{X})) \in \sigma_{\text{LA}}$ and $\theta''(\mathtt{del}(a(\bar{X}))) \notin \sigma_{\text{LA}}$, so $H'_i = \theta''(a_{\mathtt{r}}(\bar{X}, \mathtt{p}))\#id_i \in S$ and $\theta''(H_i) = \mathsf{chr}(H'_i)$. Similarly, if $A^f_i = \mathtt{del}(a(\bar{X}))$ then $H_i = a_{\mathtt{r}}(\bar{X}, N)$ and g_1 contains $N \neq \mathtt{p}$; $\theta''(\mathtt{del}(a(\bar{X}))) \in \sigma_{\text{LA}}$ and as a result $H'_i = \theta''(a_{\mathtt{r}}(\bar{X}, N'))\#id_i \in S$ with $N' = \mathtt{n}$ or $N' = \mathtt{b}$. Since N only appears in H_i and the guard $N \neq \mathtt{p}$, we can further impose that $\theta''(N) = N'$ and then $\theta''(H_i) = \mathsf{chr}(H'_i)$.

All $\theta''(A^f_i)$ are different for $1 \leq i \leq k$, and therefore, all id_i must be different. From $\mathcal{D} \models \bar{\exists}_\emptyset \theta(A^c_i)$ for $1 \leq i \leq l$ and because $\theta''(g_1) = [N_1 \neq \mathtt{p}, \ldots, N_o \neq \mathtt{p}]$ with $N_j = \mathtt{n}$ or $N_j = \mathtt{b}$ for $1 \leq j \leq o$, $\mathcal{D} \models \mathtt{true} \rightarrow \bar{\exists}_\emptyset \theta''(g_1 \wedge g_2)$. We conclude that θ'' is a ground matching substitution that matches the head with constraints from S and for which the guard is entailed.

It is not possible that $\langle \mathsf{id}(H), \epsilon, r_\rho \rangle \in T$ because $\mathsf{chr\_to\_la}$ grows monotonically, which implies that $\theta(C) = \theta''(C') \in \mathsf{chr\_to\_la}(\sigma) = \sigma_{\text{LA}}$ which contradicts with the applicability of $\theta(r)$ in σ_{LA}.

If we ignore rule priorities, all conditions are satisfied so that rule instance $\theta(r_\rho)$ can fire. The resulting state σ' has the form $\langle \theta(C), S, \mathtt{true}, T \cup \{\langle \mathsf{id}(H), \epsilon, r_\rho \rangle\} \rangle_n$. Clearly, if $\sigma_{\text{LA}} = \mathsf{chr\_to\_la}(\langle \emptyset, S, \mathtt{true}, T \rangle_n)$ and $\sigma'_{\text{LA}} = \sigma_{\text{LA}} \cup \theta(C)$ then $\sigma'_{\text{LA}} = \mathsf{chr\_to\_la}(\sigma')$. We now prove that every CHR transition firing a non-implied rule instance corresponds to a Logical Algorithms transition, also ignoring rule priorities. Both results combined give us that the priority of the highest priority rule instance is equal in both σ and σ_{LA}.

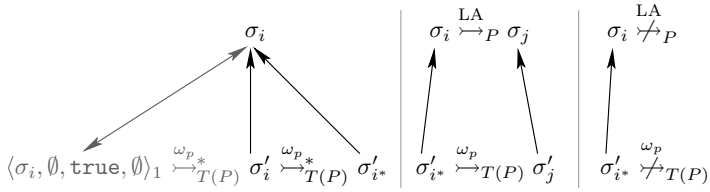


Figure 7.1: Correspondence between derivations in Logical Algorithms and CHR$^{\text{rp}}$

A transition of $\sigma = \langle \emptyset, S, \mathtt{true}, T \rangle_n$ to σ' implies that $T(P)$ contains a rule

$$p + 2 :: r_\rho \ @ \ H \implies g_1, g_2 \mid C'$$

and so the Logical Algorithms program P contains a rule

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

Let $\langle A^u, A^c \rangle = \mathsf{split}([A_1, \ldots, A_n])$ and $\theta = \mathsf{partition\_to\_mgu}(\rho, A^u)$. If $A_i = a(\bar{X}) \in A^u$ then $\theta(a_{\mathtt{r}}(\bar{X}, \mathtt{p})) \in H$. If $A_i = \mathtt{del}(a(\bar{X})) \in A^u$ then $\theta(a_{\mathtt{r}}(\bar{X}, N)) \in H$ and $(N \neq \mathtt{p}) \in g_1$. Finally, if $A_i \in A^c$ then $\theta(A_i) \in g_2$. There exists a (ground) matching substitution θ' such that $\theta'(H) \in \mathsf{chr}(S)$ and $\mathcal{D} \models \bar{\exists}_\emptyset \theta'(g_1 \wedge g_2)$.

Let $\theta'' = \theta' \circ \theta$ and let $\sigma_{\text{LA}} = \mathsf{chr\_to\_la}(\sigma)$. Because θ' is a ground substitution, $\mathcal{D} \models \bar{\exists}_\emptyset \theta'(g_1 \wedge g_2)$ implies that for all $A_i \in A^c$, $\mathcal{D} \models \theta''(A_i)$. For all positive user-defined antecedents $A_i = a(\bar{X}) \in A^u$, we have that $\theta''(a_{\mathtt{r}}(\bar{X}, \mathtt{p})) \in \mathsf{chr}(S)$ and so $\theta''(A_i) \in \sigma_{\text{LA}}$ and $\mathtt{del}(\theta''(A_i)) \notin \sigma_{\text{LA}}$. For all negative user-defined antecedents $A_i = \mathtt{del}(a(\bar{X})) \in A^u$, we have that $\theta''(a_{\mathtt{r}}(\bar{X}, N)) \in \mathsf{chr}(S)$ with $N = \mathtt{b}$ or $N = \mathtt{n}$ and so $\theta''(A_i) \in \sigma_{\text{LA}}$. We have assumed that $\theta'(r_\rho)$ is not an implied rule instance and so $\theta'(C') = \theta''(C) \not\subseteq \sigma_{\text{LA}}$.

If we again ignore rule priorities, all conditions are satisfied so that rule instance $\theta''(r)$ can fire in state σ_{LA} and it holds that $\sigma'_{\text{LA}} = \sigma_{\text{LA}} \cup \theta''(C) = \mathsf{chr\_to\_la}(\sigma')$ since $\sigma' = \langle \theta'(C'), S, \mathtt{true}, T \cup \{\langle \mathsf{id}(H), \epsilon, r_\rho \rangle\} \rangle_n$. Now we have that both the original program P and its translation $T(P)$ can fire corresponding rule instances if we ignore priorities, and so their highest priority rule instances also correspond. □

Theorem 14. *For every reachable CHR^{rp} state σ, if $\sigma \overset{\omega_p}{\underset{T(P)}{\longmapsto}} \sigma'$ then it holds that either $\mathsf{chr\_to\_la}(\sigma) = \mathsf{chr\_to\_la}(\sigma')$ or $\mathsf{chr\_to\_la}(\sigma) \overset{LA}{\underset{P}{\longmapsto}} \mathsf{chr\_to\_la}(\sigma')$.*

Proof. Implied by Lemmas 7.3.4, 7.3.6 and 7.3.8. □

Theorem 15. *For every Logical Algorithms state σ_i and reachable CHR^{rp} state σ'_i such that $\mathsf{chr\_to\_la}(\sigma'_i) = \sigma_i$, there exists a finite CHR^{rp} derivation $\sigma'_i \overset{\omega_p *}{\underset{T(P)}{\longmapsto}} \sigma'_{i*}$ for which holds that $\mathsf{chr\_to\_la}(\sigma'_{i*}) = \sigma_i$ such that if $\sigma_i \overset{LA}{\underset{P}{\longmapsto}} \sigma_j$ then $\sigma'_{i*} \overset{\omega_p}{\underset{T(P)}{\longmapsto}} \sigma'_j$ with $\mathsf{chr\_to\_la}(\sigma'_j) = \sigma_j$ and if σ_i is a final state then σ'_{i*} is also a final state.*

Proof. Implied by Lemmas 7.3.4, 7.3.6 and 7.3.8. □

Given a Logical Algorithms state σ, we can use $\langle \sigma, \emptyset, \mathtt{true}, \emptyset \rangle_1$ as initial state for the CHR$^{\text{rp}}$ derivation. Theorem 15 is illustrated by Figure 7.1.

7.3.3 Relation with Weak Bisimilarity

To capture the meaning of the above correspondence results, we relate them to the notion of (weak) bisimulation. A bisimulation is a relation between the states of a labeled transition system (LTS). A relation $R \subseteq S_1 \times S_2$ between the states in S_1 and those in S_2 is a bisimulation if $p\ R\ q$ and $p \xrightarrow{\alpha} p'$ implies that $q \xrightarrow{\alpha} q'$ with $p'\ R\ q'$, and similarly, $p\ R\ q$ and $q \xrightarrow{\alpha} q'$ implies that $p \xrightarrow{\alpha} p'$ with $p'\ R\ q'$. Here, α is the label of the transition $p \xrightarrow{\alpha} p'$ from state p to state p'. If a transition from p to p' has no observable effect, it is called a *silent* transition and denoted by $p \xrightarrow{\tau} p'$. A relation $R \subseteq S_1 \times S_2$ is a *weak* bisimulation if $p\ R\ q$ and $p \xrightarrow{\alpha} p'$ implies that $q \xrightarrow{\tau}{}^{*} q_* \xrightarrow{\alpha} q'_* \xrightarrow{\tau}{}^{*} q'$ with $p'\ R\ q'$, and vice versa with the roles of p and q swapped. Here $p \xrightarrow{\tau}{}^{*} p'$ means p and p' are linked by zero or more silent transitions.

Let S_1 be the set of valid Logical Algorithms states for program P and let $S_2 = \{\mathsf{chr\_to\_la}(\sigma) \mid \langle G, \emptyset, \mathtt{true}, \emptyset\rangle_1 \overset{\omega_p}{\rightarrowtail}{}^{*}_{T(P)} \sigma \wedge G \in S_1\}$, i.e., S_2 is found by applying the $\mathsf{chr\_to\_la}$ mapping function to all reachable CHR$^{\text{rp}}$ states for program $T(P)$. We transform the state transition systems for Logical Algorithms and CHR$^{\text{rp}}$ to labeled transition systems as follows: a Logical Algorithms transition $\sigma \overset{\text{LA}}{\rightarrowtail}_P \sigma'$ corresponds to an LTS transition $\sigma \xrightarrow{\alpha} \sigma'$ with $\alpha = \sigma' \setminus \sigma$, i.e., α represents the state change from σ to σ'. A CHR$^{\text{rp}}$ transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$ corresponds to an LTS transition $\mathsf{chr\_to\_la}(\sigma) \xrightarrow{\alpha} \mathsf{chr\_to\_la}(\sigma')$ with $\alpha = \mathsf{chr\_to\_la}(\sigma') \setminus \mathsf{chr\_to\_la}(\sigma)$ if this set is not empty and $\alpha = \tau$ otherwise.

Corollary 7.3.9. *The equality relation between the states of S_1 and S_2 is a weak bisimulation.*

7.4 Translating a subset of CHR$^{\text{rp}}$ into Logical Algorithms

In the previous section, we have shown that Logical Algorithms programs can be translated into equivalent CHR$^{\text{rp}}$ programs. In this section, we show how to do the opposite, i.e., how CHR$^{\text{rp}}$ programs can be translated into equivalent Logical Algorithms programs. This allows us to apply the meta-complexity theorem for Logical Algorithms to the translation of these CHR$^{\text{rp}}$ programs.

We need to impose some restrictions on the CHR$^{\text{rp}}$ programs that can be translated. These restrictions result from the fact that the Logical Algorithms language does not have the concept of an underlying constraint solver that offers both ask and tell built-in constraints. The following two properties are required:

1. In all *reachable* states $\sigma = \langle G, S, B, T\rangle_n$: $\mathsf{vars}(S) = \emptyset$. In words, all (stored) CHR constraints are ground.

2. All built-in constraints are comparisons; there are no built-in tell constraints.

The first property holds if the initial goal is ground and all rules are *variable restricted*, which means that all variables in the body of a rule also appear in one of the rule heads. The second property implies that all reachable states are of the form $\langle G, S, \mathtt{true}, T\rangle_n$, i.e., the built-in constraint store is always equivalent to `true`.

To simplify the presentation, we also assume that the priority of dynamic priority rules is determined by the arguments of its left-most head. In general, we can use the transformation schema given in Section 7.2.1 to ensure that the resulting Logical Algorithms rules have the correct syntactical form.

7.4.1 Translation Schema

We now show how the rules of a CHR$^{\text{rp}}$ program P are transformed into Logical Algorithms rules that form a program $T(P)$. To increase readability, we distinguish between simplification and simpagation rules on the one hand, and propagation rules on the other. A *simpagation* rule of the form

$$p :: r \ @ \ H_1, \ldots, H_{m-1} \backslash H_m, \ldots, H_n \iff g \mid B_1, \ldots, B_l$$

is transformed into

$$\begin{aligned} r' \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(H_m^{\mathsf{id}}), \ldots, \mathtt{del}(H_n^{\mathsf{id}}), \mathtt{del}(\mathtt{next\_id}(\mathit{Id}_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_l^{\mathsf{id}}, \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}} + l) \end{aligned}$$

where $H_i^{\mathsf{id}} = \mathtt{c}(\bar{X}, \mathit{Id}_i)$ if $H_i = \mathtt{c}(\bar{X})$, $B_i^{\mathsf{id}} = \mathtt{c}(\bar{X}, \mathit{Id}_{\mathsf{next}} + i - 1)$ if $B_i = \mathtt{c}(\bar{X})$ and $\mathit{Alldiff} = \{(\mathit{Id}_i \neq \mathit{Id}_j) \mid \mathcal{D} \models \bar{\exists}_{\emptyset} H_i = H_j \wedge g\}$. The disequalities in *Alldiff* are between the identifiers of those heads that are unifiable and for which the guard is still satisfiable after this unification. The case of a simplification rule is very similar. A propagation rule of the form

$$p :: r \ @ \ H_1, \ldots, H_n \implies g \mid B_1, \ldots, B_l$$

is transformed into the following two rules

$$\begin{aligned} r'_1 \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g \Rightarrow \mathtt{token}([\mathit{Id}_1, \ldots, \mathit{Id}_n], r) \\ r'_2 \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{token}([\mathit{Id}_1, \ldots, \mathit{Id}_n], r), \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(\mathtt{token}([\mathit{Id}_1, \ldots, \mathit{Id}_n], r)), \mathtt{del}(\mathtt{next\_id}(\mathit{Id}_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_l^{\mathsf{id}}, \mathtt{next\_id}(\mathit{Id}_{\mathsf{next}} + l) \end{aligned}$$

where H_i^{id}, B_i^{id} and *Alldiff* are as before. The first of these rules generates a token. This token is removed by the second rule. The tokens are needed to prevent a given rule instance from firing more than once. Note that the transformation into two rules and the use of tokens does not increase the complexity compared to the original rule, as there is only one token for each combination of rule and constraint identifiers as well as only one `next_id`/1 fact in any state.

The initial database consists of the goal, where each constraint is extended with a unique identifier, and a $\mathtt{next\_id}(\mathit{Id}_{\mathsf{next}})$ assertion, with $\mathit{Id}_{\mathsf{next}}$ the next free identifier.

Example 7.4.1 (Merge Sort)**.** *Listing 7.3 shows a* CHR$^{\text{rp}}$ *implementation of the merge sort algorithm. Its input consists of a series of n (a power of 2)* `number/1` *constraints. Its output is a sorted list of the numbers in the input, represented as* `arrow/2` *constraints, where* $\mathtt{arrow}(x, y)$ *indicates that x is right before y. The Logical Algorithms translation is shown in Listing 7.4.*

```
1 :: ms1 @ arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).
2 :: ms2 @ merge(N,A),  merge(N,B) <=> A < B | merge(2*N+1,A), arrow(A,B).
3 :: ms3 @ number(X) <=> merge(0,X).
```

Listing 7.3: A CHR$^{\text{rp}}$ implementation of merge sort

Note that in rules `ms1` *and* `ms2`*, the guard prevents the constraints matching the heads from being equal, and so there are no disequality constraints between the CHR constraint*

```
ms1′ @ 1 : arrow(X,A,Id1), arrow(X,B,Id2), A < B, next_id(NId) =>
            del(arrow(X,B,Id2)), del(next_id(NId)),
            arrow(A,B,NId), next_id(NId+1).
ms2′ @ 2 : merge(N,A,Id1), merge(N,B,Id2), A < B, next_id(NId) =>
            del(merge(N,A,Id1)), del(merge(N,B,Id2)), del(next_id(NId)),
            merge(2*N+1,A,NId), arrow(A,B,NId+1), next_id(NId+2).
ms3′ @ 3 : number(X,Id), next_id(NId) => del(number(X,Id)),
            del(next_id(NId)), merge(0,X,NId), next_id(NId+1).
```

Listing 7.4: The Logical Algorithms translation of Listing 7.3

identifiers in these rules. Using the Logical Algorithms meta-complexity result, we can derive that the total runtime of the translated merge sort algorithm is $\mathcal{O}(n \log n)$. A detailed analysis is given in Section 7.6.1 where we analyze the CHR^{rp} *implementation directly using a new meta-complexity theorem for* CHR^{rp}.

Example 7.4.2 (Less-or-Equal)**.** *We illustrate the translation of propagation rules by translating the* ***transitivity*** *rule from the* ***leq*** *program, shown below. Its translation is shown in Listing 7.5.*

```
2 :: transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

```
transitivity′1 @ 2 : leq(X,Y,Id1), leq(Y,Z,Id2), Id1 \= Id2 =>
                token([Id1,Id2],transitivity).
transitivity′2 @ 2 : leq(X,Y,Id1), leq(Y,Z,Id2), Id1 \= Id2,
        token([Id1,Id2],transitivity), next_id(NId) =>
                del(token([Id1,Id2],transitivity)), del(next_id(NId)),
                leq(X,Z,NId), next_id(NId+1).
```

Listing 7.5: LA translation of the `transitivity` rule from `leq`

Note that since in the original rule, the two heads $\texttt{leq}(\texttt{X},\texttt{Y})$ *and* $\texttt{leq}(\texttt{Y},\texttt{Z})$ *are unifiable and there is furthermore no guard to prevent them matching the same constraint, we have to add an explicit disequality between the constraint identifiers for these heads:* $\texttt{Id}_1 \neq \texttt{Id}_2$.

7.4.2 Correspondence between ω_p and LA Derivations

In this subsection, we prove that a CHR^{rp} program and its translation to Logical Algorithms are operationally equivalent. Again we introduce a mapping function:

$$\mathsf{la\_to\_chr}(\sigma) = \langle \emptyset, S, \mathtt{true}, T \rangle_n$$

where the CHR constraint store $S = \{c(\bar{X})\#Id \mid c(\bar{X}, Id) \in \sigma \wedge \mathtt{del}(c(\bar{X}, Id)) \notin \sigma\}$, the propagation history $T = \{\langle Ids, \epsilon, r\rangle \mid \mathtt{del}(\mathtt{token}(Ids, r)) \in \sigma\}$, and the next free identifier n is such that $\mathtt{next\_id}(n) \in \sigma$ and $\mathtt{del}(\mathtt{next\_id}(n)) \notin \sigma$. In the following, we consider a Logical Algorithms state σ reachable with respect to program $T(P)$ if it can be derived from an initial state consisting of CHR constraints extended with unique identifiers, and a single `next_id`/1 assertion with as argument the next free identifier. In this case, reachability amongst others implies that there can be only one (strictly) positive `next_id`/1 assertion in the database in any state, and no two CHR constraint representations share their identifier.

Theorem 16. *For every reachable LA state σ_i it holds that if $\sigma_i \overset{LA}{\rightarrowtail}_{T(P)} \sigma_j$, then either $\mathsf{la\_to\_chr}(\sigma_i) = \mathsf{la\_to\_chr}(\sigma_j)$ or there exists a finite CHR derivation $\mathsf{la\_to\_chr}(\sigma_i) = \langle \emptyset, S, \mathtt{true}, T\rangle_n \overset{\omega_p}{\rightarrowtail}_P \langle C, S', \mathtt{true}, T'\rangle_n \overset{\omega_p}{\rightarrowtail}^*_P \langle \emptyset, S'', \mathtt{true}, T'\rangle_{n'} = \mathsf{la\_to\_chr}(\sigma_j)$ consisting of an **Apply** transition, followed by zero or more **Introduce** transitions.*

Proof. Consider a transition $\sigma_i \overset{\mathrm{LA}}{\rightarrowtail}_{T(P)} \sigma_j$. The only type of transition in Logical Algorithms is the **Apply** transition which fires a rule. If $\mathsf{la\_to\_chr}(\sigma_i) = \mathsf{la\_to\_chr}(\sigma_j)$, then this rule must be of the form

$$r'_1 \ @\ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, \mathit{Alldiff}, g \Rightarrow \mathtt{token}([Id_1, \ldots, Id_m], r)$$

because all other types of rules either delete the representation of a CHR constraint which changes the CHR constraint store, or remove a token which results in an extended propagation history. We call such a rule a *token generation rule*.

Assume $\mathsf{la\_to\_chr}(\sigma_i) \neq \mathsf{la\_to\_chr}(\sigma_j)$ and the rule fired is of the form

$$\begin{aligned} r' \ @\ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(H_l^{\mathsf{id}}), \ldots, \mathtt{del}(H_m^{\mathsf{id}}), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_o^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + o) \end{aligned}$$

which corresponds to a simplification ($l = 1$) or simpagation ($l > 1$) rule. We further assume the case of a simpagation rule; the case of a simplification rule is similar. If $r' \in T(P)$ (with $l > 1$), then P contains a rule

$$p :: r \ @\ H_1, \ldots, H_{l-1} \backslash H_l, \ldots, H_m \iff g \mid B_1, \ldots, B_o$$

Since the conditions for the Logical Algorithms **Apply** transition are satisfied, there exists a ground matching substitution θ such that for each antecedent $H_i^{\mathsf{id}} = c(\bar{X}, Id_i)$ $(1 \leq i \leq m)$ it holds that $\theta(H_i^{\mathsf{id}}) \in \sigma$ and $\mathtt{del}(\theta(H_i^{\mathsf{id}})) \notin \sigma$ and so by definition of the $\mathsf{la\_to\_chr}$ function, $\theta(H_i \# Id_i) \in S$ where $\mathsf{la\_to\_chr}(\sigma_i) = \sigma'_i = \langle \emptyset, S, \mathtt{true}, T\rangle_n$. For each comparison $g_i \in g$, it holds that $\mathcal{D} \models \theta(g_i)$ and so $\mathcal{D} \models \mathtt{true} \rightarrow \bar{\exists}_\emptyset \theta(g)$. Since r is a simpagation rule, the propagation history T does not contain any element of the form $\langle \_, \_, r\rangle$. Ignoring priorities for the moment, all conditions are satisfied such that the rule instance $\theta(r)$ can fire in state σ'_i. We return to the issue of priorities further on.

After firing $\theta(r)$ in state σ'_i, the resulting state equals $\langle \theta(B_1 \wedge \ldots \wedge B_o), S', \mathtt{true}, T\rangle_n$ where $S' = S \setminus \{\theta(H_l \# Id_l), \ldots, \theta(H_m \# Id_m)\}$. In this state, the only applicable transition is the **Introduce** transition, which is applied o times before reaching a state with an empty goal. There are $o!$ possible orders in which the introductions can be applied, but the one we need is the order in which the B_i constraints appear in the rule body. Following this order, the state resulting from the introductions equals $\sigma'_j = \langle \emptyset, S'', \mathtt{true}, T\rangle_{(n+o)}$ where $S'' = S' \cup \{\theta(B_1)\#n, \ldots, \theta(B_o)\#(n + o - 1)\}$. It is easy to see that this state σ'_j equals $\mathsf{la\_to\_chr}(\sigma_j)$, the state resulting from firing Logical Algorithms rule instance $\theta(r')$ in state σ_i.

If $\mathsf{la\_to\_chr}(\sigma_i) \neq \mathsf{la\_to\_chr}(\sigma_j)$ and the rule fired is not of the form shown above, then it must have the following form

$$\begin{aligned} r'_2 \ @\ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{token}([Id_1, \ldots, Id_m], r), \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(\mathtt{token}([Id_1, \ldots, Id_m], r)), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_o^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + o) \end{aligned}$$

The corresponding CHR$^{\text{rp}}$ rule in P looks like

$$p :: r \ @ \ H_1, \ldots, H_m \implies g \mid B_1, \ldots, B_o$$

Again, since the conditions for the Logical Algorithms **Apply** transition are satisfied, there exists a ground matching substitution θ such that for each antecedent $H_i^{\mathsf{id}} = c(\bar{X}, Id_i)$ $(1 \leq i \leq m)$ in rule r_2' it holds that $\theta(H_i^{\mathsf{id}}) \in \sigma$ and $\mathtt{del}(\theta(H_i^{\mathsf{id}})) \notin \sigma$ and so by definition of the $\mathsf{la\_to\_chr}$ function, $\theta(H_i \# Id_i) \in S$ where $\mathsf{la\_to\_chr}(\sigma_i) = \sigma_i' = \langle \emptyset, S, \mathtt{true}, T \rangle_n$. For each comparison $g_i \in g$, it holds that $\mathcal{D} \models \theta(g_i)$ and so $\mathcal{D} \models \mathtt{true} \rightarrow \bar{\exists}_{\emptyset}\theta(g)$. The propagation history T cannot contain $\langle \theta([Id_1, \ldots, Id_m]), \epsilon, r \rangle$ because by definition of the $\mathsf{la\_to\_chr}$ function this would imply that the atom $\mathtt{token}(\theta([Id_1, \ldots, Id_m), r)$ was deleted in some earlier state, which contradicts with the applicability of rule instance $\theta(r_2')$. If we again ignore the issue of priorities, all conditions are satisfied such that $\theta(r)$ can fire in state σ_i'.

After firing $\theta(r)$ in state σ_i', the resulting state equals $\langle \theta(B_1 \wedge \ldots \wedge B_o), S, \mathtt{true}, T' \rangle_n$ where $T' = T \cup \{\langle [Id_1, \ldots, Id_m], \epsilon, r \rangle\}$. In this state, the only applicable transition is the **Introduce** transition, which is applied o times before reaching a state with an empty goal. Given again that these introductions are applied in the order in which the B_i constraints appear in the rule body, then the resulting state equals $\sigma_j' = \langle \emptyset, S', \mathtt{true}, T' \rangle_{(n+o)}$ where $S' = S \cup \{\theta(B_1)\#n, \ldots, \theta(B_o)\#(n+o-1)\}$. It is again easy to see that this state σ_j' equals $\mathsf{la\_to\_chr}(\sigma_j)$, the state resulting from firing Logical Algorithms rule instance $\theta(r_2')$ in state σ_i.

This proves the theorem if we ignore priorities. Theorem 17 (see next) shows that each CHR$^{\text{rp}}$ rule firing has a corresponding Logical Algorithms rule firing. Under the assumption that this theorem also holds ignoring rule priorities, we have that the highest priority rule instances are the same in both programs given corresponding states and ignoring token generation rules. □

Theorem 17. *For every reachable CHR^{rp} state σ_i and reachable Logical Algorithms state σ_i' with $\mathsf{la\_to\_chr}(\sigma_i') = \sigma_i$, there exists a finite Logical Algorithms derivation $\sigma_i' \overset{LA}{\longmapsto}{}^*_{T(P)} \sigma_{i^*}'$ with $\mathsf{la\_to\_chr}(\sigma_{i^*}') = \sigma_i$ such that if $\sigma_i = \langle \emptyset, S, \mathtt{true}, T \rangle_n \overset{\omega_p}{\longmapsto}_P \langle C, S', \mathtt{true}, T' \rangle_n \overset{\omega_p}{\longmapsto}{}^*_P \langle \emptyset, S'', \mathtt{true}, T' \rangle_{n'} = \sigma_j$ where the derivation consists of a single **Apply** transition, followed by zero or more **Introduce** transitions, then $\sigma_{i^*}' \overset{LA}{\longmapsto}_{T(P)} \sigma_j'$ with $\mathsf{la\_to\_chr}(\sigma_j') = \sigma_j$ and if σ_i is a final state then σ_{i^*}' is also a final state.*

Proof. Let there be given a reachable LA state σ_i' with $\mathsf{la\_to\_chr}(\sigma_i') = \sigma_i$. Because of Theorem 16, state σ_i is also reachable in CHR$^{\text{rp}}$ with respect to program P. Assume $\sigma_i = \langle \emptyset, S, \mathtt{true}, T \rangle_n \overset{\omega_p}{\longmapsto}_P \langle C, S', \mathtt{true}, T' \rangle_n \overset{\omega_p}{\longmapsto}{}^*_P \langle \emptyset, S'', \mathtt{true}, T' \rangle_{n'} = \sigma_j$ where the derivation consists of a single **Apply** transition, followed by zero or more **Introduce** transitions, and let $\theta(r)$ be the CHR$^{\text{rp}}$ rule instance that fired in state σ_i. If r is simplification $(l = 1)$ or simpagation $(l > 1)$ rule

$$p :: r \ @ \ H_1, \ldots, H_{l-1} \backslash H_l, \ldots, H_m \iff g \mid B_1, \ldots, B_o$$

then $\theta(H_i)\#id_i \in S$ for $1 \leq i \leq m$ with $id_i \neq id_j$ if $i \neq j$, and $\mathcal{D} \models \bar{\exists}_{\emptyset}\theta(g)$. Furthermore, $T(P)$ contains a rule

$$\begin{aligned} r' \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_m^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \\ \mathtt{del}(H_l^{\mathsf{id}}), \ldots, \mathtt{del}(H_m^{\mathsf{id}}), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ B_1^{\mathsf{id}}, \ldots, B_o^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + o) \end{aligned}$$

Now let θ' be a ground matching substitution such that $\theta'|_{\mathsf{vars}(\theta)} = \theta$ where $\theta'|_{\mathsf{vars}(\theta)}$ is the projection of θ' on the variables in θ, and such that both $\theta'(Id_i) = id_i$ for $1 \leq i \leq m$ and $\theta'(Id_{\mathsf{next}}) = n$. Since for $1 \leq i \leq m$, $H_i^{\mathsf{id}} = c(\bar{X}, Id_i)$ if $H_i = c(\bar{X})$, it holds that $\theta'(H_i^{\mathsf{id}}) \in \sigma_i'$ and $\mathtt{del}(\theta'(H_i^{\mathsf{id}})) \notin \sigma_i'$. Also, $\mathcal{D} \models \bar{\exists}_{\emptyset}\theta(g)$ implies $\mathcal{D} \models \theta(g_i)$ for each comparison $g_i \in g$.[7] The *Alldiff* conditions hold because $\theta'(Id_i) = \theta'(Id_j)$ implies that $i = j$. Because of the reachability of state σ_i', there is exactly one strictly positive $\mathtt{next\_id}/1$ assertion in σ_i' whose argument equals n. Finally, the rule conclusion cannot be already included in the state σ_i' because it includes amongst others the deletion of at least one of the antecedents. Ignoring priorities, all conditions are satisfied such that rule instance $\theta'(r')$ can fire in state σ_i', resulting in a state $\sigma_j' = \mathsf{la\_to\_chr}(\sigma_j)$. As stated earlier in the proof of Theorem 16, the combination of Theorems 16 and 17 without taking into account the priorities, implies that the highest priority applicable rule instances are the same in corresponding states, ignoring token generation rules.

Now assume that in the CHR$^{\mathrm{rp}}$ state σ_i, a rule instance $\theta(r)$ fires where r is a propagation rule:

$$p :: r \ @ \ H_1, \ldots, H_m \implies g \mid B_1, \ldots, B_o$$

In this case the Logical Algorithms translation $T(P)$ contains the following rules:

$$\begin{aligned} r_1' \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g \Rightarrow \ & \mathtt{token}([Id_1, \ldots, Id_n], r) \\ r_2' \ @ \ p : H_1^{\mathsf{id}}, \ldots, H_n^{\mathsf{id}}, \mathit{Alldiff}, g, \mathtt{token}([Id_1, \ldots, Id_n], r), \mathtt{next\_id}(Id_{\mathsf{next}}) \Rightarrow \ & \mathtt{del}(\mathtt{token}([Id_1, \ldots, Id_n], r)), \mathtt{del}(\mathtt{next\_id}(Id_{\mathsf{next}})), \\ & B_1^{\mathsf{id}}, \ldots, B_l^{\mathsf{id}}, \mathtt{next\_id}(Id_{\mathsf{next}} + l) \end{aligned}$$

A similar analysis as above shows that there exists a matching substitution θ' with $\theta'|_{\mathsf{vars}(\theta)} = \theta$ and both $\theta'(Id_i) = id_i$ for $1 \leq i \leq m$ and $\theta'(Id_{\mathsf{next}}) = n$, such that rule instance $\theta'(r_1')$ can fire (ignoring priorities) if $\mathtt{token}([id_1, \ldots, id_n], r) \notin \sigma_i'$ and $\theta'(r_2')$ otherwise. If $\theta'(r_1')$ fires then the resulting state σ_{i^*}' equals $\sigma_i' \cup \{\mathtt{token}([id_1, \ldots, id_n], r)\}$ and clearly $\mathsf{la\_to\_chr}(\sigma_{i^*}') = \mathsf{la\_to\_chr}(\sigma_i')$. Moreover, in state σ_{i^*}', rule instance $\theta'(r_2')$ can fire and for the resulting state σ_j' it holds that $\mathsf{la\_to\_chr}(\sigma_j') = \sigma_j$. If already $\mathtt{token}([id_1, \ldots, id_n], r) \in \sigma_i'$ then the same reasoning holds with $\sigma_i' = \sigma_{i^*}'$.

Finally, assume that CHR$^{\mathrm{rp}}$ state σ_i is a final state. If σ_i' is not a final Logical Algorithms state, then because of Theorem 16, the only applicable rules are those that do not change the result of the $\mathsf{la\_to\_chr}$ function. Only the token generation rules satisfy this property. Since they only generate tokens and these tokens do not appear in their antecedents, these rules can fire only finitely many times before a final Logical Algorithms state σ_{i^*} is reached. □

7.5 Implementing CHR$^{\mathrm{rp}}$, the Logical Algorithms Way

This section presents a new implementation for CHR$^{\mathrm{rp}}$, based on the implementation proposal for Logical Algorithms by Ganzinger and McAllester [2002], as well as on the scheduling algorithm presented in [De Koninck, 2007]. The purpose of this implementation is not to replace our existing CHR$^{\mathrm{rp}}$ implementation as presented in Chapter 4, but to support a new meta-complexity theorem for CHR$^{\mathrm{rp}}$, based on the result for Logical Algorithms, and extended towards the full CHR$^{\mathrm{rp}}$ language. This includes in particular support for non-ground constraints and a built-in constraint theory. We note that a better worst case complexity for certain operations is not always worthwhile in practice due to

[7] Because $\theta(g)$ is ground, there is no existential quantification.

larger constant factors in the average case. Also, the proposed implementation may not always achieve a better complexity than the existing implementation. The main purpose remains to have a relatively straightforward way to derive for a given CHR^{rp} program, a bound that is guaranteed to be an upper-bound for at least the implementation proposed. Since the meta-complexity result is insensitive to constant factors, we can present the new implementation as a source-to-source transformation to regular CHR.

The proposed implementation consists of the compilation of the CHR^{rp} rules of the input program into regular CHR rules in which matching is made explicit, combined with a scheduler module that is responsible for the execution control. The implementation is correct if it is executed according to the refined operational semantics of CHR. We have based our implementation on the high-level implementation proposal for Logical Algorithms by Ganzinger and McAllester [2002], extended where necessary to support general built-in constraints. By using a CHR implementation with advanced indexing support, such as the K.U.Leuven CHR system [Schrijvers and Demoen, 2004], our implementation also offers strong complexity guarantees that facilitate a new meta-complexity theorem for CHR^{rp}, similar to the one for Logical Algorithms (see Section 7.6). In the following, we make use of Prolog as CHR's host language, but the implementation can easily be adapted to work with a different host language.

7.5.1 Overview

The implementation is based on a form of lazy (on-demand) matching with retainment of previously computed partial matches. It combines the concept of alpha and beta memories from the RETE algorithm [Forgy, 1982], with lazy matching as for example implemented by the LEAPS algorithm [Miranker et al., 1990].[8] The basic idea is as follows. A new constraint can function both as a single headed partial or full match, and as an extension of an existing partial match into either a new (larger) partial match or a full match. In order to extend partial matches, all previously computed matches are stored. A scheduler decides which partial match is extended with which constraint, or which full match has its corresponding rule instance fired. More details on the scheduler are given in Section 7.5.3.

First, to simplify the presentation, we propose an alternative syntax for CHR^{rp} rules. An *intermediate form* CHR^{rp} rule looks as follows:

$$p :: r \ @ \ s_1 A_1, \ldots, s_n A_n \iff B$$

where $s_i \in \{+, -, ?\}$ and A_i is an atom for $1 \leq i \leq n$. If $s_i = +$ or $s_i = -$ then A_i must be a CHR constraint and if $s_i =?$ then A_i must be a built-in constraint. An intermediate form CHR^{rp} rule corresponds to a regular CHR^{rp} rule as follows: a term $+A$ corresponds to a kept head A, a term $-A$ corresponds to a removed head A, and a term $?A$ corresponds to a conjunct of the rule guard. The main advantage of the intermediate form is that it supports specifying a join order for the heads, as well as an evaluation order for the guards. In particular it supports specifying the evaluation of part of the guard after having computed only a partial rule match. The intermediate form gives us the same syntactical flexibility as exists in the Logical Algorithms language where comparisons are interleaved with the (kept and removed) user-defined antecedents.

Consider, in general, a simpagation rule of the form

$$p :: r \ @ \ H_1, \ldots, H_i \backslash H_{i+1}, \ldots, H_n \iff g \mid B$$

[8] Most current CHR systems, including the K.U.Leuven CHR system and the CHR^{rp} system use a variant of the LEAPS algorithm for rule matching.

where the guard g is a conjunction of atomic guards $g_1, \ldots, g_m$. We can rewrite this rule in intermediate form syntax (amongst others) as follows:

$$p :: r \ @ \ +H_1, \ldots, +H_i, -H_{i+1}, \ldots, -H_n, ?g_1, \ldots, ?g_m \iff B$$

In the following, we assume that all rules have the following form

$$p :: r \ @ \ \pm H_1, ?g_1, \pm H_2, ?g_2, \ldots, \pm H_n, ?g_n \iff B$$

where $\pm$ means $+$ or $-$. Each g_i $(1 \leq i \leq n)$ can be a conjunction of primitive built-in constraints, and can in particular also be equal to `true`. The transformation from regular CHR$^{\text{rp}}$ syntax to intermediate form syntax can be done automatically using the above transformation schema, or by hand.

Using terminology similar to that of Ganzinger and McAllester [2002], we refer to a partial match, matching the heads $H_1, \ldots, H_i$ and satisfying the partial guard $g_1 \wedge \ldots \wedge g_{i-1}$, as a *suspended strong prefix firing*. If also the partial guard g_i is satisfied, we speak of a *regular* (or *non-suspended*) strong prefix firing. A constraint matching the next head H_{i+1} is called a *prefix extension* of such a (regular) strong prefix firing. A prefix firing that consists of all heads is (also) called a (suspended or regular) *rule firing*.[9] Every prefix firing contains the left-most head and hence determines the rule priority. In our implementation, we assume that all guards are monotone, i.e., once they are entailed by the built-in constraint store, they remain entailed in any later state. This is in fact required by the CHR operational (and declarative) semantics, although most current CHR systems also support non-monotone (impure) guards like for example `var`/1 in CHR on top of Prolog.

7.5.2 Program-Dependent Part

The program-dependent part of our implementation (i.e., the part that depends on the actual program to be implemented) consists of rules for

- generating a representation for CHR constraint occurrences and deleting them when the represented constraint is removed;
- generating and scheduling constraints representing prefix firings, prefix extensions and rule firings and deleting them when a constituent constraint is removed;
- matching prefix firings with prefix extensions, firing rule instances, and managing suspended prefix and rule firings.

The different types of rules of the program-dependent part are illustrated by using a running example program, namely Dijkstra's shortest path algorithm, already given in the Logical Algorithms language in Example 7.2.2 and given here in CHR$^{\text{rp}}$ intermediate form syntax. To illustrate non-trivial head matching, we have added a rule `d1` that removes simple loops from the input graph.

```
    1 :: d1 @ -e(V,_,V),   ?true                          <=> true.
    1 :: d2 @ +source(V),  ?true                          <=> dist(V,0).
    1 :: d3 @ -dist(V,D1), ?true, +dist(V,D2), ?(D2 < D1) <=> true.
D + 2 :: d4 @ +dist(V,D),  ?true, +e(V,C,U),   ?true      <=> dist(U,D+C).
```

[9] A rule firing actually means a rule instance that is applicable. To avoid confusion, we refer to the actual firing of such a rule firing as *firing a rule instance*.

Constraint Occurrence Representation

Although CHR^{rp} constraints and CHR constraints obviously have the same syntax and semantics (i.e., multi-set semantics with non-monotone deletion), we introduce a new representation for them to allow unambiguous reference, reduce work in case of constraint reactivation, and support the efficient deletion of those prefix firings, prefix extensions, and rule firings in which they participate (see further). For each CHR^{rp} constraint of predicate c/n, we create a set of unique *occurrence representations* $c\_\texttt{occ}\_i/(n+1)$, one for each occurrence of the predicate in a rule head. The arguments of a $c\_\texttt{occ}\_i/(n+1)$ constraint consist of the arguments of the original c/n constraint, together with a unique constraint identifier that is shared by all occurrence representations. This identifier is an uninstantiated variable as long as the constraint is in the store and is instantiated the moment that the constraint is to be deleted. For each user-defined constraint predicate c/n with m occurrences, the occurrence representations are generated using rules of the following form.

c(X$_1$,...,Xn) <=> c_occ_1(X$_1$,...,Xn,Id), ..., c_occ_m(X$_1$,...,Xn,Id).

For the example program, these rules look as follows.

```
source(V) <=> source_occ_1(V,Id).
dist(V,D) <=> dist_occ_1(V,D,Id), dist_occ_2(V,D,Id), dist_occ_3(V,D,Id).
e(V,C,U)  <=> e_occ_1(V,C,U,Id),  e_occ_2(V,C,U,Id).
```

RETE Memory Constraints

Regular and suspended prefix firings as well as prefix extensions are represented as CHR constraints. We call them RETE memory constraints because they coincide with the alpha and beta memories of the RETE algorithm. The RETE memory constraints contain all arguments of their constituent CHR constraints, as well as their identifiers. Each RETE memory constraint moreover has its own unique identifier. We use the following functors for RETE memory constraints:

- r_pf_i for a regular (non-suspended) prefix firing of rule r, consisting of i heads, and r_pf_i_suspended for its suspended version
- r_pe_i for a prefix extension, consisting of the $(i+1)^{\text{th}}$ head of rule r
- r_rf for a (regular) rule firing of rule r and r_rf_suspended for its suspended version.

If in a rule r, the partial guard after the i^{th} head equals `true`, then there is no suspended version of the i-headed prefix firings of r, or of its rule firings if r is an i-headed rule. In the example program, the following prefix firings, prefix extensions and rule firings are defined:

- d1_rf/4
- d2_rf/3
- d3_pf_1/4, d3_pe_1/3, d3_rf/6 and d3_rf_suspended/6
- d4_pf_1/4, d4_pe_1/4 and d4_rf/7

Suspended Prefix and Rule Firings

Suspended prefix and rule firings are converted into regular prefix and rule firings as soon as the relevant part of the guard is entailed. If on the other hand this partial guard is disentailed, the suspended prefix or rule firing is removed. Given a rule in intermediate form syntax

$$p :: r \ @ \ \pm H_1, ?g_1, \pm H_2, ?g_2, \ldots, \pm H_n, ?g_n \iff B$$

we generate the following rules:

- For each i-headed suspended prefix firing:

  ```
  r_pf_i_suspended(X1,...,Xm,Id1,...,Idi,SId) <=>
              gi | r_pf_i(X1,...,Xm,Id1,...,Idi,SId),
              schedule_pf(r_i(Y1,...,Yl),p,SId).
  r_pf_i_suspended(X1,...,Xm,Id1,...,Idi,SId) <=> \+ gi | true.
  ```

  where $\mathtt{Y}_1, \ldots, \mathtt{Y}_l$ are those variables in $\mathtt{X}_1, \ldots, \mathtt{X}_m$ that also appear in H_{i+1}

- For each rule firing:

  ```
  r_rf_suspended(X1,...,Xm,Id1,...,Idn,SId) <=> gn |
              r_rf(X1,...,Xm,Id1,...,Idn,SId), schedule_rf(p,SId).
  r_rf_suspended(X1,...,Xm,Id1,...,Idn,SId) <=> \+ gn | true.
  ```

Note that if g_i or g_n equals `true`, then we can apply unfolding to replace occurrences of respectively `r_pf_`i`_suspended`$/(m+i+1)$ and `r_rf_suspended`$/(m+n+1)$ by the bodies of the corresponding rules above (see [Tacchella et al., 2007]). After this unfolding step, some of the above rules may be removed. In the example program, only a rule firing of rule `d3` can be suspended.

```
d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=>
        D2 < D1 | d3_rf(V,D1,D2,Id1,Id2,SId), schedule_rf(1,SId).
d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=> \+ (D2 < D1) | true.
```

In each second rule, `\+` C is a safe approximation of the negation of constraint C, i.e., it is only entailed if constraint C cannot possibly hold. In the Prolog context, the built-in negation as failure can be used.

Suspended constraints are attached to all guarded variables so that they are reactivated whenever one of these variables is affected by a built-in constraint. We assume that both attaching and detaching can be done in constant time, although certain current CHR implementations like the K.U.Leuven CHR system do not support detaching in constant time.

Scheduling

Each constraint occurrence corresponds to a (potentially suspended) rule firing if it is the only head of a single-headed rule, a (potentially suspended) prefix firing if it is the first head of a multi-headed rule, and a prefix extension in all other cases. A conversion between constraint occurrence and rule firing, prefix firing or prefix extension is made as soon as the constraint in question matches with the head. If such a match is shown to be

impossible, the constraint occurrence is discarded. Let there be given a head constraint $c(X_1, \ldots, X_n)$. The following function is used to construct a head match.

$$\mathsf{head\_match}([X|\bar{X}]) = \begin{cases} \langle [X|\bar{Y}], g\rangle \text{ if } X \text{ is a variable and } X \notin \mathsf{vars}(\bar{X}) \\ \langle [Y|\bar{Y}], (Y = X) \wedge g\rangle \text{ otherwise} \end{cases}$$
$$\text{where } \langle \bar{Y}, g\rangle = \mathsf{head\_match}(\bar{X})$$
$$\mathsf{head\_match}(\epsilon) = \langle \epsilon, \mathtt{true}\rangle$$

Now, for each rule in intermediate form syntax

$$p :: r \ @ \ \pm H_1, ?g_1, \pm H_2, ?g_2, \ldots, \pm H_n, ?g_n \iff B$$

and for $1 \leq i \leq n$ we generate the rules below where $H_i = c(X'_1, \ldots, X'_n)$ is the j^{th} occurrence of the user-defined constraint predicate c/n, $\langle [X_1, \ldots, X_n], g\rangle = \mathsf{head\_match}([X'_1, \ldots, X'_n])$, and $\{Y_1, \ldots, Y_m\} = \mathsf{vars}(H_i) \setminus \mathsf{vars}(\{H_1, \ldots, H_{i-1}\})$.

- If $i = n = 1$:

  c_occ_j(X1,...,Xn,Id) <=> g | r_rf_suspended(Y1,...,Ym,Id,SId).
  c_occ_j(X1,...,Xn,Id) <=> \+ g | true.

- If $i = 1$ and $n > 1$:

  c_occ_j(X1,...,Xn,Id) <=> g | r_pf_1_suspended(Y1,...,Ym,Id,SId).
  c_occ_j(X1,...,Xn,Id) <=> \+ g | true.

- Otherwise, if $i > 1$:

  c_occ_j(X1,...,Xn,Id) <=> g | r_pe_$i-1$(Y1,...,Ym,Id,SId),
  schedule_pe(r_i-1(Z1,...,Zl),SId).
  c_occ_j(X1,...,Xn,Id) <=> \+ g | true.

  where $\{Z_1, \ldots, Z_l\} = \mathsf{vars}(H_i) \cap \mathsf{vars}(\{H_1, \ldots, H_{i-1}\})$.

In the above, if $g = \mathtt{true}$ then the second rule of each pair of rules can be discarded. The suspended prefix and rule firings can sometimes be replaced with regular prefix and rule firings by unfolding.

In the example program, only the first occurrence of the e/3 constraint has a non-trivial head match (the first and last argument must be the same). All single-headed prefix and rule firings are followed by the trivial guard `true` and so we only generate regular prefix and rule firings. They are scheduled using the `schedule_pf`/3 and `schedule_rf`/2 predicates.

```
source_occ_1(V,Id) <=> d2_rf(V,Id,SId), schedule_rf(1,SId).

dist_occ_1(V,D,Id) <=> d3_pf_1(V,D,Id,SId), schedule_pf(d3_1(V),1,SId).
dist_occ_2(V,D,Id) <=> d3_pe_1(D,Id,SId), schedule_pe(d3_1(V),SId).
dist_occ_3(V,D,Id) <=> d4_pf_1(V,D,Id,SId), schedule_pf(d4_1(V),D+2,SId).

e_occ_1(V,C,U,Id) <=> V = U | d1_rf(V,C,Id,SId), schedule_rf(1,SId).
e_occ_1(V,C,U,Id) <=> \+ (V = U) | true.
e_occ_2(V,C,U,Id) <=> d4_pe_1(C,U,Id,SId), schedule_pe(d3_1(V),SId).
```

Prefix firings and extensions are scheduled using a key containing their shared variables. For example for the prefix firings consisting of the first head of rule `d3` and the corresponding prefix extensions consisting of the second head of the same rule, the key equals `d3_1(V)`.

Similar to the suspended prefix and rule firings, the constraint occurrences are attached to all guarded variables. We again assume that both attaching and detaching can be done in constant time.

Matching and Firing

The scheduler initiates the firing of a rule instance by asserting a `fire`/1 constraint, and the matching of a prefix firing with a prefix extension by asserting a `match`/2 constraint. These constraints have as arguments the identifiers of the corresponding RETE memory constraints. After matching a prefix firing with a prefix extension, a new suspended prefix or rule firing is generated. For a given n-headed rule r with $n > 1$ and for $1 \leq i \leq n-2$, we generate the following rule

```
r_pf_i(X1,...,Xm,Id1,...,Idi,SId1), r_pe_i(Xm+1,...,Xl,Idi+1,SId2) \
        match(SId1,SId2) <=> Idi+1 \== Id1, ..., Idi+1 \== Idi |
                r_pf_i+1_suspended(X1,...,Xl,Id1,...,Idi+1).
```

and similarly for $i = n-1$:

```
r_pf_n-1(X1,...,Xm,Id1,...,Idn-1,SId1), r_pe_n-1(Xm+1,...,Xl,Idn,SId2) \
        match(SId1,SId2) <=> Idn \== Id1, ..., Idn \== Idn-1 |
                r_rf_suspended(X1,...,Xl,Id1,...,Idn).
```

A rule firing of an n-headed rule r with body B is fired as follows:

```
r_rf_i(X1,...,Xm,Id1,...,Idn,SId), fire(SId) <=>
                Idr(1) = dead, ..., Idr(l) = dead, B.
```

where $r(1), \ldots, r(l)$ are the indices of the removed heads of the rule (if any). We furthermore add the following rules at the end of the code, to make sure the CHR compiler detects that the `match`/2 and `fire`/1 constraints are never to be stored.

```
match(_,_) <=> true.
fire(_) <=> true.
```

For the example program, the generated code is as follows:

```
d1_rf(V,C,Id,SId), fire(SId) <=> Id = dead.
d2_rf(V,Id,SId), fire(SId) <=> dist(V,0).
d3_pf_1(V,D1,Id1,SId1), d3_pe_1(D2,Id2,SId2) \ match(SId1,SId2) <=>
        Id2 \== Id1 | d3_rf_suspended(V,D1,D2,Id1,Id2,SId).
d3_rf(V,D1,D2,Id1,Id2,SId), fire(SId) <=> Id1 = dead.
d4_pf_1(V,D,Id1,SId1), d4_pe_1(C,U,Id2,SId2) \ match(SId1,SId2) <=>
        Id2 \== Id1 | d4_pf(V,D,C,U,Id1,Id2,SId), schedule_rf(D+2,SId).
d4_rf(V,D,C,U,Id1,Id2,SId), fire(SId) <=> dist(U,D+C).

match(_,_) <=> true.
fire(_) <=> true.
```

Clean-up

Whenever a constraint's identifier variable is instantiated, its occurrence representations, as well as those RETE memory constraints in which it participates, are removed. The rules look as follows.

- For the i^{th} occurrence representation for constraint predicate c/n:

  c_occ_i(X$_1$,...,X$_n$,Id) <=> nonvar(Id) | true.

- For an i-headed suspended prefix firing of rule r:

  r_pf_i_suspended(X$_1$,...,X$_m$,Id$_1$,...,Id$_i$,SId) <=> nonvar(Id$_1$) | true.
  ...
  r_pf_i_suspended(X$_1$,...,X$_m$,Id$_1$,...,Id$_i$,SId) <=> nonvar(Id$_i$) | true.

- For an i-headed regular prefix firing of rule r:

  r_pf_i(X$_1$,...,X$_m$,Id$_1$,...,Id$_i$,SId) <=> nonvar(Id$_1$) | remove_pf(SId).
  ...
  r_pf_i(X$_1$,...,X$_m$,Id$_1$,...,Id$_i$,SId) <=> nonvar(Id$_i$) | remove_pf(SId).

- For a prefix extension of an i-headed prefix firing of rule r:

  r_pe_i(X$_1$,...,X$_m$,Id,SId) <=> nonvar(Id) | remove_pe(SId).

- For a suspended rule firing of an n-headed rule r:

  r_rf_suspended(X$_1$,...,X$_m$,Id$_1$,...,Id$_n$,SId) <=> nonvar(Id$_1$) | true.
  ...
  r_rf_suspended(X$_1$,...,X$_m$,Id$_1$,...,Id$_n$,SId) <=> nonvar(Id$_n$) | true.

- For a regular rule firing of an n-headed rule r:

  r_rf(X$_1$,...,X$_m$,Id$_1$,...,Id$_n$,SId) <=> nonvar(Id$_1$) | remove_rf(SId).
  ...
  r_rf(X$_1$,...,X$_m$,Id$_1$,...,Id$_n$,SId) <=> nonvar(Id$_n$) | remove_rf(SId).

The predicates `remove_pf`/1, `remove_pe`/1 and `remove_rf`/1 remove respectively a prefix firing, prefix extension and rule firing from the schedule. The following clean-up rules are generated for the example program.

```
source_occ_1(V,Id) <=> nonvar(Id) | true.
dist_occ_1(V,D,Id) <=> nonvar(Id) | true.
dist_occ_2(V,D,Id) <=> nonvar(Id) | true.
dist_occ_3(V,D,Id) <=> nonvar(Id) | true.
e_occ_1(V,C,U,Id)  <=> nonvar(Id) | true.
e_occ_2(V,C,U,Id)  <=> nonvar(Id) | true.

d1_rf(V,C,Id,SId)          <=> nonvar(Id)  | remove_rf(SId).
d2_rf(V,Id,SId)            <=> nonvar(Id)  | remove_rf(SId).
d3_pf_1(V,D1,Id1,SId)      <=> nonvar(Id1) | remove_pf(SId).
d3_pe_1(D2,Id2,SId)        <=> nonvar(Id2) | remove_pe(SId).
d3_rf(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id1) | remove_rf(SId).
d3_rf(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id2) | remove_rf(SId).
d4_pf_1(V,D,Id1,SId)       <=> nonvar(Id1) | remove_pf(SId).
d4_pe_1(C,U,Id2,SId)       <=> nonvar(Id2) | remove_pe(SId).
d4_rf(V,D,C,U,Id1,Id2,SId) <=> nonvar(Id1) | remove_rf(SId).
```

```
d4_rf(V,D,C,U,Id1,Id2,SId) <=> nonvar(Id2) | remove_rf(SId).

d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id1) | true.
d3_rf_suspended(V,D1,D2,Id1,Id2,SId) <=> nonvar(Id2) | true.
```

7.5.3 Program-Independent Part: The Scheduler

The scheduler implements the `schedule_rf`/2, `schedule_pf`/3 and `schedule_pe`/2 predicates. It furthermore implements the `execute`/0 predicate which retrieves and executes the highest priority scheduled task. This task either is the firing of a rule instance by asserting a `fire`/1 constraint, or the matching of a prefix firing with a prefix extension by asserting a `match`/2 constraint. The `execute`/0 predicate recursively calls itself until no more tasks are scheduled. It is first called after processing the initial goal.

For the implementation of the scheduler, we use a variant of the scheduling algorithm presented in [De Koninck, 2007]. In this work, we present a data structure representing *schedules* which are sets of *elements* and *nodes*. It supports the following operations: creating a new schedule, adding a new element or a new node to a given schedule, deleting an element or a node from its schedule, merging two schedules (after which the resulting schedule contains the elements and nodes of both original schedules), and finally, generating a new *match*. A match is the combination of an element and a node, both of which belong to the same schedule. The algorithm ensures that no match is generated twice. Furthermore, it only fails to generate a new match if all elements and nodes belonging to any single schedule have already been matched. The algorithm ensures that all defined operations take quasi-constant time.

We can use our algorithm to maintain which prefix firings are still to match with which prefix extensions. Here, a prefix firing is mapped to a schedule's element, and a prefix extension is mapped to a schedule's node. A schedule of [De Koninck, 2007] roughly corresponds to an $\mathcal{W}(r,t)$ data structure of [Ganzinger and McAllester, 2002]. These $\mathcal{W}(r,t)$ data structures consist of a series (implemented as a linear linked list) of *prefix blocks*, which are sets of prefix firings and (apart from the last one) are associated with a prefix extension.

The semantics of the $\mathcal{W}(r,t)$ data structure is that the prefix firings of a given prefix block are still to match with the prefix extension associated to it, as well as with all prefix extensions associated to subsequent prefix blocks. The last prefix block has no associated prefix extension, and represents those prefix firings that have been matched with all prefix extensions and hence are passive (or *completed* using the terminology of Ganzinger and McAllester [2002]). Whenever a prefix extension is deleted, its prefix block is merged with the next prefix block.

There is one $\mathcal{W}(r,t)$ data structure for each prefix length of each rule and for each combination of arguments shared between a prefix firing and prefix extension. Each prefix block is represented as a (local) priority queue whose items are the block's prefix firing. The highest priority item of each prefix block, together with its associated prefix extension, is also represented in a global priority queue. This prefix block representative is updated whenever the highest priority prefix firing of the prefix block is removed, a new prefix firing has the highest priority, or the associated prefix extension is removed. The global priority queue furthermore contains a representative for each rule firing. The reason for using two layers of priority queues is to reduce the amount of work needed when the prefix firings of a prefix block all become passive due to a prefix extension removal. It is the global priority queue that determines the next task to perform, i.e., matching a prefix firing with

a prefix extension, or firing a rule instance.

In the context of CHR^{rp}, built-in constraint (in particular equality constraints) on the arguments shared between a prefix firing and extension, may require merging of $\mathcal{W}(r,t)$ data structures. The data structure of De Koninck [2007] supports schedule merges in quasi constant time. The most notable difference with the $\mathcal{W}(r,t)$ data structure of Ganzinger and McAllester [2002] is that the prefix blocks form a circular linked list. Using this representation, merging schedules consists of cross-linking the circular lists and reactivating the prefix firings that were passive before the merge. Special care is taken to prevent both that a prefix firing is being matched with the same prefix extension more than once, and that a prefix firing 'misses' a prefix extension.

One consequence of using a circular linked list instead of a linear one to represent the prefix blocks, is that it is unclear (or more precisely, too expensive to decide) which prefix firings become passive whenever a prefix extension is deleted. Therefore, this decision is postponed until the scheduler tries to match the prefix firing with the next prefix extension in line. For complexity reasons, it is important that all prefix firings that have simultaneously been reactivated, and have not been matched with a prefix extension since this reactivation, are simultaneously made passive in time independent of the number of prefix firings affected. In [De Koninck, 2007], a so-called *element schedule* based on a stack is proposed to supports this. In our context, we need an element schedule that is based on priority queues. It works as follows.

We use three types of priority queues. The first one is a single *global* priority queue which contains an item for each rule firing, for each *active* prefix firing that either has not been passive before or has been matched with at least one prefix extension since its last activation, and finally, for each set of prefix firings that have been simultaneously activated and have not been matched with a prefix extension since. A second type of priority queues is called a *local* queue and represents the above mentioned sets of prefix firings. Finally, the third type of queues is the *passive* queue which contains an item for each passive (completed) prefix firing. There is one passive queue for each schedule. Essentially, we again use two layers of priority queues. Whenever a set of previously passive prefix firings, represented as a passive priority queue, is reactivated because of a new prefix extension or because of a schedule merge, this passive priority queue becomes a local priority queue and has a representative inserted into the global priority queue. If such a representative is the highest priority item in the global priority queue, and an `execute`/0 call is made, then the highest priority prefix firing of the represented local queue is removed and dealt with as an ordinary prefix firing. The representatives of local priority queues are updated (and potentially removed) similarly to how this is done in the $\mathcal{W}(r,t)$ data structure of [Ganzinger and McAllester, 2002].

Example 7.5.1. *Figure 7.2 illustrates the prefix blocks, the different types of priority queues, and their contents. The global queue, which is shared by all schedules, contains the rule firings RF_1 and RF_2, the prefix firings PF_1, PF_4, PF_5 and PF_8 (the last of which belongs to another schedule), and the local queue representative LQ_1. The represented local queue contains the prefix firings PF_2 and PF_3 which are by definition also in the same prefix block. The schedule's passive queue contains the prefix firings PF_6 and PF_7. The schedule has two prefix blocks, which are associated with respectively the prefix extensions PE_1 and PE_2.*

Using our approach, the cost of deleting items from the global priority queue can be amortized to one of the following events: a new rule firing, a new prefix firing, a new prefix extension (for each representative of a local priority queue), or a match between a prefix

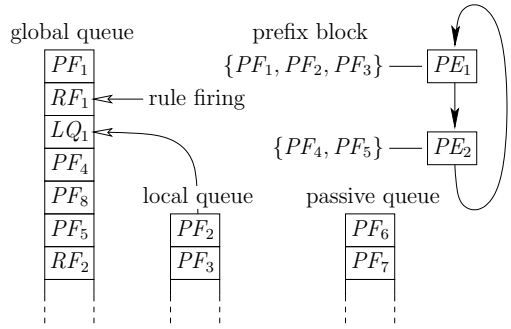


Figure 7.2: Example schedule with global, local and passive priority queues

firing and a prefix extension (which corresponds to either a new larger prefix firing, or a rule firing).

In [Ganzinger and McAllester, 2002], retrieving the schedule for a given prefix firing or prefix extension is done by hashing. In our approach, we use a variant of hashing, which we call *non-ground hashing* and which consists of first replacing all variables by a unique identifier, and then using the resulting (ground) term for hashing. Unifications may require rehashing the affected keys and potentially also the merging of schedules.

7.5.4 Priority Queues

A priority queue or heap is a data structure that contains a set of prioritized items and supports the following operations: inserting and removing an item, finding a highest priority item and merging with another queue. The implementation proposal by Ganzinger and McAllester [2002] suggests the use of two types of priority queues, one for the fixed priorities, where each of the supported operations takes constant time, and Fibonacci heaps for the dynamic priorities.

Fibonacci heaps [Fredman and Tarjan, 1987] are a type of priority queue that offer $\mathcal{O}(1)$ amortized time insertion, heap merging and finding a highest priority item, and $\mathcal{O}(\log n)$ amortized time item removal with n the number of items in the queue. It is suggested by Ganzinger and McAllester [2002] that by using only one node per priority, using linked lists to represent the items that share this priority, the item removal cost can be reduced to $\mathcal{O}(\log N)$ with N the number of distinct priorities. However, this increases the cost of heap merging from $\mathcal{O}(1)$ for a single merge operation to a total cost of $\mathcal{O}(n \log N)$ for merging heaps when there are n items in total and N distinct priorities. A CHR implementation of Fibonacci heaps is described by Sneyers et al. [2006a]. It can easily be extended to support multiple heaps that can be merged and to use only one node for each distinct priority per heap.

7.6 A New Meta-Complexity Result for $\mathrm{CHR}^{\mathrm{rp}}$

In this section, we give a new meta-complexity result for $\mathrm{CHR}^{\mathrm{rp}}$. It extends the result via translation to Logical Algorithms, by also supporting built-in constraints and non-ground CHR constraints. In the following, we assume that hash tables support $\mathcal{O}(1)$ insertion, removal, and retrieval of all elements that match a given (ground) key. This assumption is also made by Ganzinger and McAllester [2002] and holds on average as long as the hash

function is good enough. Our scheduling data structure of [De Koninck, 2007] makes use of the union-find algorithm, which results in a factor $\alpha(n)$ in the complexity of its operations, where α is the inverse of the Ackermann function. Since this inverse is positive and less than 5 for all practical values of n, we ignore this factor in our complexity result.

We start by looking at the complexity of the different operations supported by our scheduler.

Lemma 7.6.1 (Scheduler Costs)**.** *Let N be the number of distinct priorities, and assume that a priority queue merge takes some abstract time T, then the schedule operations have the following amortized cost:*

- $\mathcal{O}(1)$ *and* $\mathcal{O}(\log N)$ *for each* `schedule_pf/3`, `remove_pf/1`, `remove_pe/1`, `remove_rf/1` *and* `execute/0` *operation involving respectively a static and dynamic priority rule*
- $\mathcal{O}(T+1)$ *and* $\mathcal{O}(T+\log N)$ *for each* `schedule_pe/2` *operation involving respectively a static and dynamic priority rule*
- $\mathcal{O}(1)$ *for each schedule merge and* `schedule_rf/2` *operation*

Proof. We only consider the costs related to the priority queue operations. The other costs are shown to be (quasi) constant in [De Koninck, 2007]. We now look at the different operations in detail:

- A `schedule_pf`/3 call consists of inserting the new prefix firing into the global priority queue. We also account to this event, the cost of making the new prefix instance passive the first time. That operation consists of a removal from the global priority queue and an insertion into the schedule's passive queue. The total cost is $\mathcal{O}(1)$ if the element has a static priority, and $\mathcal{O}(\log N)$ if it has a dynamic priority.
- A `schedule_pe`/2 call requires the insertion of a new representative for the local priority queue of reactivated prefix firings, into the global priority queue. We also take into account here, the cost of making all the reactivated prefix firings passive that have not been matched with a prefix extension since the reactivation. That operation consists of removing the representative and merging the local priority queue with the schedule's passive queue. The cost is $\mathcal{O}(T+1)$ for a static priority rule and $\mathcal{O}(T+\log N)$ time for a dynamic priority one.
- A `schedule_rf`/2 call requires an insertion into the global priority queue which takes $\mathcal{O}(1)$ time.
- A `remove_pf`/1 call consists of deleting the prefix firing from the global priority queue, from a local priority queue or from a passive queue. A deletion from a local queue may moreover require an update of the global queue (removal and insertion). In total, this takes $\mathcal{O}(1)$ time for a static priority rule and $\mathcal{O}(\log N)$ time for a dynamic priority rule.
- A `remove_pe`/1 call does not require any priority queue operations, and so the cost is $\mathcal{O}(1)$.
- A `remove_rf`/1 call requires a removal from the global priority queue which takes $\mathcal{O}(1)$ time if it involves a static priority rule and $\mathcal{O}(\log N)$ time if it involves a dynamic priority rule.

- An `execute`/0 call requires retrieval and potential removal (if the retrieved item corresponds to a rule firing, or to a prefix firing that becomes passive) of the highest priority item in the global priority queue. If the retrieved item represents a prefix firing or set of prefix firings that need to be made passive, the cost of this operation is already accounted for by a previous `schedule_pf`/3 or `schedule_pe`/2 operation. In such case, we call the `execute`/0 call *unsuccessful*. An unsuccessful `execute`/0 call is followed by another `execute`/0 call until either such a call is successful, or the global priority queue is empty and thus a final state is reached. The cost of all unsuccessful `execute`/0 calls can be amortized to previous events. If in case of a successful `execute`/0 call, the item retrieved from the global priority queue corresponds to the representative of a local priority queue, the operation requires a removal of the highest priority item (prefix firing) from this local queue, an insertion of the prefix firing into the global priority queue, and potentially the insertion of a new representative for the local queue into the global queue. The cost of a successful `execute`/0 call therefore equals $\mathcal{O}(1)$ if it involves a static priority rule and $\mathcal{O}(\log N)$ otherwise.
- A schedule merge requires the reactivation of the passive prefix firings of the merged schedules. The cost analysis is similar to that of a `schedule_pe`/2 call. Moreover, each schedule merge can be accounted for by at least one `schedule_pf`/3 or `schedule_pe`/2 call as the resulting schedule contains at least one prefix firing or extension more than each of the original schedules, and so the number of schedule merges is bounded by the number of prefix firings and extensions. Therefore, the cost of a single schedule merge can be considered constant.

□

In the above lemma, we have made abstraction of the cost of priority queue merge operations. Such merges take place when the prefix firings in a local priority queue all become passive. In such an event, the local priority queue is merged with the schedule's passive queue. It is easy to see that the cost of merging priority queues for static priorities takes constant time per merge operation. In Section 7.5 a bound is given on the total cost of merging Fibonacci heaps with one node per distinct priority, given the number of items ever inserted into the heaps. The following lemma makes use of this result.

Lemma 7.6.2 (Fibonacci Heap Merging Cost)**.** *The total cost of Fibonacci heap merges is* $\mathcal{O}((P_d + A_d) \cdot \log N)$ *where* P_d *is the number of strong prefix firings of dynamic priority rules,* A_d *is the number of constraints that may participate in a dynamic priority rule instance, and* N *is the number of distinct rule priorities.*

Proof. We count the number of items ever inserted into the local and passive Fibonacci heaps, and then apply the result of Section 7.5. A local priority queue basically is the same as a passive priority queue in which items are no longer inserted. Therefore, a merge between a local queue and a passive queue can be seen as a special case of a merge between two passive queues and so we only need to consider these passive priority queues. Each item inserted in such a queue is either a prefix firing that has never been passive before, or a prefix firing that has been matched with a prefix extension at least once since its last activation. The total number of these items is $\mathcal{O}(P_\mathsf{d} + A_\mathsf{d})$ because each prefix firing that has been matched with a prefix extension is by definition a strong prefix firing, and each new prefix firing either results from matching a (smaller) strong prefix firing and extension and hence corresponds to a (potentially suspended) strong prefix firing, or consists of a

single head in which case it corresponds to a constraint assertion. Now given the number of items ever inserted into the passive priority queues, the total cost of merging Fibonacci heaps hence is $\mathcal{O}((P_\mathsf{d} + A_\mathsf{d}) \cdot \log N))$. □

We are now ready to formulate the new meta-complexity theorem.

Theorem 18. *Let A_s and A_d be the number of assertions of constraints with an occurrence in respectively a static and dynamic priority rule. Let P_s and P_d be the number of strong prefix firings of respectively static and dynamic priority rules. The time complexity of a CHR^{rp} program executed using our implementation is*

$$\mathcal{O}(C_{ask} \cdot (A_s + P_s + (A_d + P_d) \cdot \log N) + B \cdot C_{tell} \cdot (K + C_{ask} \cdot S))$$

where N is the number of distinct priorities, C_{ask} is the cost of evaluating a built-in ask constraint, C_{tell} is the cost of solving a built-in tell constraint, and B is the number of built-in tell constraints asserted in rule bodies; K is the maximum number of distinct combinations (keys) of arguments shared between prefix firings and extensions in which any given variable occurs, and S is the maximum number of suspended strong prefix firings (i.e., those that are followed by a non-trivial guard) and suspended instances of constraint occurrences (i.e., whose arguments are not mutually distinct variables) in which any given variable occurs.

Proof. Each new CHR constraint causes the creation of constraint occurrences which are converted into RETE memory constraints as soon as the implicit guard on the constraint arguments is entailed (i.e., the constraint matches the head in question). These RETE memory constraints are scheduled using `schedule_pf`/3 for the single-headed prefix firings, `schedule_rf`/2 for the single-headed rule firings, and `schedule_pe`/2 for the prefix extensions. The total cost of these operations, including the cost of priority queue merges (for the `schedule_pe`/2 calls), equals $\mathcal{O}(C_\mathsf{ask} \cdot (A_\mathsf{s} + (A_\mathsf{d} + P_\mathsf{d}) \log N))$. Each constraint deletion causes the deletion of those RETE memory constraints in which the deleted constraint participated. The total cost related to deletion therefore is $\mathcal{O}(A_\mathsf{s} + P_\mathsf{s} + (A_\mathsf{d} + P_\mathsf{d}) \log N)$. Each prefix firing is inserted into its schedule at most once and hence it can also be removed from this schedule only once (when one of its constituent constraints is removed). Those prefix firings that consist of at least two heads, correspond to a strong prefix firing as they are generated at a priority higher than or equal to that of the highest priority rule firing. Thus, using Lemma 7.6.1 and including the cost of checking the relevant parts of the guard, the cost for inserting (and deleting) these prefix firings is $\mathcal{O}(C_\mathsf{ask} \cdot (P_\mathsf{s} + P_\mathsf{d} \log N))$.

A built-in tell constraint is processed as follows. The keys used to identify the schedules and that are affected by the built-in constraint, are rehashed. If the built-in constraint causes two or more schedules to have the same key, these schedules are merged. The cost of rehashing is proportional to the number of affected keys and the cost of a schedule merge is constant by Lemma 7.6.1. A built-in constraint moreover requires the reactivation of the suspended prefix firings and rule firings, as well as those constraint occurrences for which it is not decided whether they match with the corresponding head or not. The reactivated prefix and rule firings have their guard checked and are potentially scheduled as regular (non-suspended) prefix and rule firings. The reactivated constraint occurrences also have their (implicit) guard checked, and are potentially scheduled as single-headed prefix firings, single-headed rule firings, or prefix extensions. The cost of the scheduling operations was already taken into account above. The remaining cost per built-in tell constraint is $\mathcal{O}(C_\mathsf{tell} \cdot (K + C_\mathsf{ask} \cdot S))$. □

Because the values of S and K might be difficult to determine in practice, we propose the following bounds: $S = \mathcal{O}(A_{\mathsf{s}} + A_{\mathsf{d}} + P_{\mathsf{s}} + P_{\mathsf{d}})$ and $K = \mathcal{O}(A_{\mathsf{s}} + A_{\mathsf{d}})$. We have used the cost of solving a built-in tell constraint as an upper-bound on the number of variables that are affected. Note that in absence of built-in constraints, the theorem given here is essentially the same as the one for Logical Algorithms.[10]

7.6.1 Examples

We illustrate the meta-complexity theorem on some examples, and compare with the results obtained by using the approach of Frühwirth [2002b].

Example 7.6.3 (Less-or-Equal)**.** *A first example is the* `leq` *program, given in* CHR$^{\text{rp}}$ *by Listing 4.1. We use a slightly different priority assignment compared to that version to simplify the analysis. In particular, we have given the* `idempotence` *rule a higher priority.*

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

Given an initial goal consisting of n `leq`*/2 constraints where the arguments are taken from a set of n distinct variables, we derive the following values for the parameters:*

- P_{s}*: the number of strong prefix firings is $\mathcal{O}(n^2)$ for the* `idempotence` *rule, $\mathcal{O}(n)$ for the* `reflexivity` *rule, $\mathcal{O}(n^2)$ for the* `antisymmetry` *rule, and $\mathcal{O}(n^3)$ for the* `transitivity` *rule. These numbers are found by looking at the degrees of freedom for each constraint occurrence, based on the domain of the arguments, and given those arguments that are already fixed by the left-most heads. For example for the* `transitivity` *rule, we know that there are $\mathcal{O}(n^2)$ constraints matching the first head, and $\mathcal{O}(n)$ constraints matching the second head, given the first. Our reasoning is based on the fact that at priority 2 and lower, all* `leq`*/2 constraints have set semantics because of the* `idempotence` *rule.*[11]

- A_{s}*: the number of* `leq`*/2 constraints asserted is $\mathcal{O}(n^3)$ (by the* `transitivity` *rule).*

- B*: the number of built-in constraints is bounded by the number of rule firings of the* `antisymmetry` *rule, and hence is $\mathcal{O}(n^2)$.*

- K*: the schedule keys are the combination of* `X` *and* `Y` *in both the* `antisymmetry` *rule and the* `idempotence` *rule, and* `Y` *in the* `transitivity` *rule. There are at most $\mathcal{O}(n)$ different keys in which any given variable occurs.*

- S*: for any variable, and in a state in which a built-in constraint can be asserted, there are up to $\mathcal{O}(n)$ suspended instances of the* `leq`*/2 occurrence in the* `reflexivity` *rule. There can be no suspended prefix or rule firings.*

- C_{ask} *and* C_{tell}*: the cost of evaluating a built-in ask constraint and the cost of solving a built-in tell constraint is constant (at least for the given query pattern).*

[10] The Logical Algorithms result makes use of the size of the initial database instead of the number of assertions. We have that $A_{\mathsf{s}} = \mathcal{O}(|\sigma_0| + P_{\mathsf{s}} + P_{\mathsf{d}})$.

[11] We assume here that the `idempotence` rule always removes the most recently asserted duplicate.

Filling in these parameters in the formula given by Theorem 18 gives us a worst case time complexity of

$$\mathcal{O}(1 \cdot (n^2 + n^3) + (0 + 0) \cdot \log 1) + n^2 \cdot 1 \cdot (n + 1 \cdot n)) = \mathcal{O}(n^3)$$

This corresponds to the actual worst-case complexity for an initial goal of the form

$$\{\texttt{leq}(X_1, X_2), \ldots, \texttt{leq}(X_{n-1}, X_n), \texttt{leq}(X_n, X_1)\}$$

The approach of Frühwirth [2002b] does not apply since the `transitivity` *rule is a propagation rule and hence no suitable ranking function can be found.*

Example 7.6.4 (Merge Sort)**.** *Consider the* CHR$^{\mathrm{rp}}$ *implementation of the merge sort algorithm, first given in Example 7.4.1 (Section 7.4) and repeated here for easy reference.*

```
1 :: ms1 @ arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).
2 :: ms2 @ merge(N,A),  merge(N,B) <=> A < B | merge(2*N+1,A), arrow(A,B).
3 :: ms3 @ number(X) <=> merge(0,X).
```

We show that the total runtime of the algorithm is $\mathcal{O}(n \log n)$ given an initial goal consisting of n `number/1` *constraints.*

No new `number/1` *constraints are ever asserted. Rule* `ms3` *converts one* `number/1` *constraint into one* `merge/2` *constraint each time it fires. The number of (strong) prefix firings for rule* `ms3` *hence is $\mathcal{O}(n)$. Rule* `ms2` *decreases the number of* `merge/2` *constraints by one and so it can fire $n-1$ times. In any state, there are at most two* `merge/2` *constraints with the same first argument. This invariant holds in the initial state because there are no* `merge/2` *constraints in the initial goal and rule* `ms2` *can fire after each new* `merge/2` *constraint assertion, enforcing the invariant. Because of the invariant, the number of prefix firings for rule* `ms2` *is limited to $\mathcal{O}(n)$.*

Using similar reasoning it holds that in any state, there are at most two `arrow/2` *constraints in the store with the same first argument. Now we define that in a given state, two numbers x_1 and x_m are connected by a* chain *of length $m-1$ if the constraint store contains $\texttt{arrow}(x_1, x_2), \texttt{arrow}(x_2, x_3), \ldots, \texttt{arrow}(x_{m-1}, x_m)$. At priority 2 it holds that for each $\texttt{merge}(m, x)$ constraint in the store, the maximal length of a chain starting in x is m. Indeed, this holds for the initial $\texttt{merge}(0, \_)$ constraints and if it holds for $\texttt{merge}(m, \_)$ constraints, it also holds for $\texttt{merge}(2 \cdot m + 1, \_)$ constraints, because when such a constraint is asserted, two chains of length m are linked with an extra* `arrow/2` *constraint and merged by up to $2 \cdot m$ firings of rule* `ms1`*. Two $\texttt{merge}(m, \_)$ constraints are combined into a $\texttt{merge}(2 \cdot m + 1, \_)$ constraint, so the n $\texttt{merge}(0, \_)$ constraints asserted by rule* `ms3` *are replaced by $n/2$ $\texttt{merge}(1, \_)$ constraints, which in turn are combined into $n/4$ $\texttt{merge}(3, \_)$ constraints and so on until finally 1 $\texttt{merge}(n-1, \_)$ constraint remains. The sum of all m in these $\texttt{merge}(m, \_)$ constraints is $\mathcal{O}(n \log n)$. Rule* `ms1` *fires $\mathcal{O}(m)$ times after every new $\texttt{merge}(m, \_)$ constraint assertion and because there are at most two* `arrow/3` *constraints with the same first argument, there are $\mathcal{O}(n \log n)$ strong prefix firings of rule* `ms1`*.*

In conclusion, for an input database consisting of n `number/1` *constraints, there are $\mathcal{O}(n \log n)$ strong prefix firings for rule* `ms1`*, $\mathcal{O}(n)$ for rule* `ms2` *and $\mathcal{O}(n)$ for rule* `ms3`*. Using the meta-complexity theorem, which simplifies to the one for Logical Algorithms because there are no built-in tell constraints, the total runtime is $\mathcal{O}(n \log n)$, which is also a tight complexity bound. We now compare this result with the result found by using the meta-complexity theorem of Frühwirth [2002b].*

Using a similar analysis as above, we can derive that $D = \mathcal{O}(n \log n)$ and $c_{\mathsf{max}} = \mathcal{O}(n)$ where n is the number of `number/1` *constraints in the query.*[12] *The cost of head matching (O_{H_r}), guard checking (O_{G_r}), adding built-in constraints (O_{C_r}), and adding and removing CHR constraints (O_{B_r}), can all be assumed constant. The number of heads n_r of a rule $r \in P$ is at most 2. Filling in these numbers, we derive a total worst case complexity of $\mathcal{O}(n^3 \log n)$, which is clearly suboptimal.*

7.6.2 Comparison with the LA Meta-Complexity Result

In this subsection, we show that our translation from Logical Algorithms to CHR$^{\mathrm{rp}}$ of Section 7.3, combined with the CHR$^{\mathrm{rp}}$ implementation presented in Section 7.5, satisfies the complexity requirements needed for the Logical Algorithms meta-complexity result to hold. We assume here that the comparison antecedents in Logical Algorithms programs are scheduled after the corresponding user-defined antecedents in the translation, and that the guards on the mode indicators (these have the form $N \neq \mathtt{p}$) are scheduled right after the head to which they apply.

Theorem 19. *The time complexity of Logical Algorithms programs executed by first translating them into CHR^{rp} programs using the translation schema of Section 7.3, and then executing the resulting CHR^{rp} program using the implementation of Section 7.5, is $\mathcal{O}(|\sigma_0| + P_s + (P_d + A_d) \cdot \log N)$ with σ_0, P_s, P_d, A_d and N as defined in Section 7.2.1.*

Proof. The translation of a Logical Algorithms program P consists of two parts as defined in Section 7.3. The first part, denoted by $T_{S+D}(P)$, contains for each user-defined predicate a/n the following rules:

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

It is easy to see that for an initial goal containing no constraints of the form $a_{\mathtt{r}}(\bar{X}, M)$ and since these are the only rules that assert such a constraint, in any state it holds that if $a_{\mathtt{r}}(\bar{X}, M_1)\#i_1$ and $a_{\mathtt{r}}(\bar{X}, M_2)\#i_2$ are in the CHR constraint store, then $i_1 = i_2$ and $M_1 = M_2$. This implies that the number of strong prefix firings for these rules is bounded by the number of assertions of $a(\bar{X})$ or $\mathtt{del}(a(\bar{X}))$.

The second part of the translation, denoted by $T_R(P)$, contains for each Logical Algorithms rule

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

a set of rules

$$p + 2 :: r_\rho \mathbin{@} H \implies g_1, g_2 \mid C'$$

as shown in the translation schema of Section 7.3.1. Amongst these rules is one, say $r_{\rho'}$, with a maximal number of heads, namely as many as there are user-defined antecedents in

[12] In Theorem 4.2 of [Frühwirth, 2002b] a worst case upper-bound of $c_{\mathsf{max}} = \mathcal{O}(c + D)$ is used, with c the number of constraints in the query, which becomes $c_{\mathsf{max}} = \mathcal{O}(n \log n)$ in this example. The bound we use is tight, i.e., $c_{\mathsf{max}} = \Theta(n)$.

$A_1, \ldots, A_n$. Because the (implicit and explicit) guards on the mode indicators of the head constraints are scheduled as soon as they are decidable, and because the comparisons are scheduled at corresponding places, it is easy to see that the number of strong prefix firings of rule $r_{\rho'}$ is the same as the number of strong prefix firings of Logical Algorithms rule r. The other r_ρ rules are restricted versions of $r_{\rho'}$ and therefore have at most as many strong prefix firings as $r_{\rho'}$.

The assertions with occurrences in dynamic priority rules are of the form $a_{\mathtt{r}}(\bar{X}, \_)$. The set and deletion semantics rules ensure that the number of these assertions is the same in the original program and in its translation. Now using our new meta-complexity result for CHR$^{\text{rp}}$ (Theorem 18), we derive that the total runtime complexity of the translated program is $\mathcal{O}(|\sigma_0| + P_{\mathsf{s}} + (P_{\mathsf{d}} + A_{\mathsf{d}}) \cdot \log N)$.[13] □

7.6.3 Comparison with the "As Time Goes By" Approach

In Section 7.2.2 we already briefly compared the LA meta-complexity result with the theorem given by Frühwirth [2002b]. In this subsection, we make the comparison complete by also considering built-in constraints, using the new meta-complexity theorem presented in Section 7.6.

Let there be given a CHR$^{\text{rp}}$ program P in which each rule has the same (static) priority. Theorem 3 in Section 4.3.3 states that such a CHR$^{\text{rp}}$ program and its corresponding CHR program (which is found by removing the rule priorities) have the same derivations. Therefore, such programs are suitable for comparing the result of [Frühwirth, 2002b] with the result of Theorem 18 in Section 7.6. In Section 7.2.2 we have already shown that the number of strong prefix firings is $\mathcal{O}\left(D \cdot \sum_{r \in P} c_{\mathsf{max}}^{n_r}\right)$ where D is the derivation length (i.e., the number of rule firings), and c_{max} is the maximal number of CHR constraints in the store in any state. The number of constraint assertions is $\mathcal{O}(c_{\mathsf{max}} + D)$. If we assume that the initial goal does not contain any built-in constraints (as is done in [Frühwirth, 2002b]), then the number of built-in constraints is $\mathcal{O}(D)$. The number of suspended prefix firings is bounded by $\mathcal{O}\left(\sum_{r \in P} c_{\mathsf{max}}^{n_r}\right)$ in any state and the number of suspended assertions by $\mathcal{O}(c_{\mathsf{max}})$. Now, filling in these parameters in the CHR$^{\text{rp}}$ meta-complexity result gives us that the total runtime complexity is

$$\mathcal{O}\left(O_C \cdot D \sum_{r \in P} (c_{max}^{n_r} \cdot O_{G_r})\right) \tag{7.3}$$

where $O_C = \sum_{r \in P}(O_{C_r})$. This formula strongly resembles the result of Frühwirth [2002b] which, assuming the cost of head matching O_{H_r} and adding and removing CHR constraints O_{B_r} is constant, equals

$$\mathcal{O}\left(D \sum_{r \in P} (c_{max}^{n_r} \cdot O_{G_r} + O_{C_r})\right) \tag{7.4}$$

The difference lies in how built-in tell constraints are dealt with. In our CHR$^{\text{rp}}$ implementation, as well as in any CHR implementation based on the refined operational semantics of CHR, a built-in tell constraint causes the constraints or matches whose variables are affected, to be reconsidered.[14] Because each individual (atomic) built-in constraint is dealt with separately, this may cost more in total than the naive approach taken in [Frühwirth, 2002b] in which after each rule firing, *all* constraints or matches are reconsidered *once*.

[13] Since $A_{\mathsf{s}} = \mathcal{O}(|\sigma_0| + P_{\mathsf{s}} + P_{\mathsf{d}})$.

[14] Which constraints are reactivated depends on the wake-up policy used for the **Solve** transition, see also [Schrijvers, 2005, Section 5.4.2].

So, while in certain rather exceptional cases, a naive approach to dealing with built-in tell constraints might in fact be better than the usual approach of selective reactivation (as can be seen by comparing Formulas (7.3) and (7.4)), in general we expect the latter approach to be an improvement over the naive one. Moreover, in these exceptional cases, the meta-complexity theorem of [Frühwirth, 2002b] does not apply to optimized CHR implementations such as the K.U.Leuven CHR system, i.e., in these cases it does not overestimate the actual worst case time complexity.

7.7 Conclusions

In this chapter, we have investigated the relationship between the Logical Algorithms language and Constraint Handling Rules. We have presented an elegant translation schema from Logical Algorithms to CHR$^{\text{rp}}$. The original program and its translation are shown to be essentially weakly bisimilar. However, our current CHR$^{\text{rp}}$ system (see Chapter 4) does not give the complexity guarantees needed for the Logical Algorithms meta-complexity theorem to hold via this translation.

As a first step towards applying the Logical Algorithms meta-complexity result to CHR$^{\text{rp}}$ programs, we have shown how a subclass of CHR$^{\text{rp}}$ can be translated into Logical Algorithms. By using this translation, we can directly apply the meta-complexity theorem for Logical Algorithms to the translated CHR$^{\text{rp}}$ programs. A drawback is that the CHR$^{\text{rp}}$ programs that can be translated this way, are restricted to those that do not make use of an underlying constraint solver.

In order to remedy both the limitation that the translation from Logical Algorithms to CHR$^{\text{rp}}$ does not exhibit the required complexity when executing translated Logical Algorithms programs using our CHR$^{\text{rp}}$ system, and the restriction of those CHR$^{\text{rp}}$ programs that can be translated to Logical Algorithms and hence to which the Logical Algorithms meta-complexity result can be applied, we have proposed a new implementation for the complete CHR$^{\text{rp}}$ language that gives strong complexity guarantees. The implementation is based on the high-level implementation proposal of Ganzinger and McAllester [2002] as well as on the scheduling data structure of De Koninck [2007], and consists of the compilation of CHR$^{\text{rp}}$ rules into (regular) CHR rules, combined with a scheduler that controls the execution. The implementation supports a new and accurate meta-complexity theorem for CHR$^{\text{rp}}$. When combining the translation from Logical Algorithms to CHR$^{\text{rp}}$ with the new implementation, the new meta-complexity theorem implies the Logical Algorithms meta-complexity result. Moreover, it is shown that in general[15] the new theorem is at least as accurate as the meta-complexity result for CHR given by Frühwirth [2002b]. This is illustrated on two non-trivial examples, one of which contains both built-in constraints and propagation rules and therefore cannot be analyzed using the Logical Algorithms approach or Frühwirth's result.

7.7.1 Related Work

The time complexity of programs is in general expressed in terms of the number of elementary operations, e.g., the number of logical inferences in Prolog, function applications in a functional programming language, or rule applications in a language such as CHR. However, while in most languages, these elementary operations all take constant time,[16]

[15] Apart from some rather exceptional cases, see Section 7.6.3.

[16] Prolog unification takes more than constant time in general, but not under certain restrictions such as those imposed in Mercury [Somogyi et al., 1996].

this is not the case in a language like CHR where each rule application results from a complex matching phase.

In this work, we have made a mapping from the number of elementary operations (like prefix and rule firings or constraint assertions) to time complexity. To the best of our knowledge, and apart from the results in [McAllester, 1999, Ganzinger and McAllester, 2001, 2002] and [Frühwirth, 2002a,b], there is no other work with a similar goal. There are many other formalisms though in which elementary operations take more than constant time. One such formalism is term rewriting, as implemented by the Maude system [Clavel et al., 1999] or the ACD term rewriting language [Duck et al., 2006]. It is known that AC matching, which is used by most of these languages, is NP-complete. Another formalism is that of production rule systems like Drools [Proctor et al., 2007] or Jess [Friedman-Hill, 2007]. Production rules are in many ways similar to Constraint Handling Rules. However unlike CHR, these systems are not often used as general purpose programming language, and therefore, algorithmic complexity has never been much of a concern. More work exists on the derivation of the number of elementary operations. In the context of CHR, this mostly concerns the number of rule firings, which is often derived as part of termination analysis [Frühwirth, 2000, Pilozzi et al., 2007, Voets et al., 2007].

Another related topic is that of space complexity, an issue that is not dealt with in this chapter. In the context of CHR, the memory reuse techniques developed by Sneyers et al. [2006b] are crucial to achieve optimal space complexity as is shown in [Sneyers et al., 2008]. The latter also introduces a space complexity meta-theorem for CHR, stating that the space complexity is $\mathcal{O}(D + p)$ where D is the derivation length and p is the number of propagation rule firings (which takes into account the size of the propagation history).

Bibliography

Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 1997.

Henning Christiansen. CHR grammars. *Theory and Practice of Logic Programming*, 5 (4-5):467–501, 2005.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. The Maude system. In Paliath Narendran and Michaël Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer, 1999.

Leslie De Koninck. Mergeable schedules for lazy matching. Technical Report CW 505, Department of Computer Science, K.U.Leuven, 2007.

Cinzia Di Giusto, Maurizio Gabbrielli, and Maria Chiara Meo. Expressiveness of multiple heads in CHR. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *35th Conference on Current Trends in Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 759–763. Springer, 2009.

Gregory J. Duck, Peter J. Stuckey, and Sebastian Brand. ACD term rewriting. In Sandro Etalle and Miroslaw Truszczynski, editors, *22nd International Conference on Logic Pro-*

gramming, volume 4079 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 2006.

Gregory J. Duck, Peter J. Stuckey, and Martin Sulzmann. Observable confluence for Constraint Handling Rules. In Veronica Dahl and Ilkka Niemelä, editors, *23rd International Conference on Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2007.

Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

Ernest Friedman-Hill. *JESS 7.0p2: The rule engine for the Java platform*, 2007. http://herzberg.ca.sandia.gov/jess.

Thom Frühwirth. Proving termination of constraint solver programs. In Krzysztof R. Apt, Antonis C. Kakas, Eric Monfroy, and Francesca Rossi, editors, *Joint ERCIM/Compulog Net Workshop on New Trends in Constraints: Selected papers*, volume 1865 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 2000.

Thom Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *8th International Conference on Principles of Knowledge Representation and Reasoning*, pages 547–557. Morgan Kaufmann, 2002a.

Thom Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In *Quantitative Aspects of Programming Languages: Selected Papers*, volume 59 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002b.

Maurizio Gabbrielli, Jacopo Mauro, and Maria Chiara Meo. On the expressive power of priorities in CHR. In *11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 267–276. ACM, 2009.

Maurizio Gabbrielli, Jacopo Mauro, Maria Chiara Meo, and Jon Sneyers. Decidability properties for fragments of chr. *Theory and Practice of Logic Programming*, 10(4-6): 611–626, 2010.

Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *1st International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2001.

Harald Ganzinger and David A. McAllester. Logical algorithms. In Peter J. Stuckey, editor, *18th International Conference on Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2002.

Christian Holzbaur and Thom Frühwirth. Constraint Handling Rules reference manual, release 2.2. Technical Report TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, 1998.

David A. McAllester. On the complexity analysis of static analyses. In Agostino Cortesi and Gilberto Filé, editors, *6th International Symposium on Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 312–329. Springer, 1999.

Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *8th National Conference on Artificial Intelligence*, pages 685–692. AAAI Press / The MIT Press, 1990.

Paolo Pilozzi, Tom Schrijvers, and Danny De Schreye. Proving termination of CHR in Prolog: A transformational approach. In Dieter Hofbauer and Alexander Serebrenik, editors, *9th International Workshop on Termination*, pages 30–33, 2007.

Mark Proctor, Michael Neale, Michael Frandsen, Sam Griffith, Jr, Edson Tirelli, Fernando Meyer, and Kris Verlaenen. *Drools Documentation, Version 4.0.3*, 2007. http://www.jboss.com/products/rules.

Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, 2005.

Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *1st Workshop on Constraint Handling Rules: Selected Contributions*, volume 2004-01 of *Ulmer Informatik-Berichte*, pages 1–5. Universität Ulm, 2004.

Tom Schrijvers and Thom Frühwirth. Optimal union-find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.

Jon Sneyers. Turing-complete subclasses of CHR. In Maria Garcia de la Banda and Enrico Pontelli, editors, *24th International Conference on Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 759–763. Springer, 2008.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In Michael Fink, Hans Tompits, and Stefan Woltran, editors, *20th Workshop on Logic Programming*, INFSYS Research Report 1843-06-02, pages 182–191. TU Wien, 2006a.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. Memory reuse for CHR. In Sandro Etalle and Miroslaw Truszczynski, editors, *22nd International Conference on Logic Programming*, volume 4079 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2006b.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. To appear in ACM Transactions on Programming Languages and Systems, 2008.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

Paolo Tacchella, Maria Chiara Meo, and Maurizio Gabbrielli. Unfolding in CHR. In Michael Leuschel and Andreas Podelski, editors, *9th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 179–186. ACM, 2007.

Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In Tom Schrijvers and Thom Frühwirth, editors, *3rd Workshop on Constraint Handling Rules*, Report CW 452, pages 125–140. Department of Computer Science, K.U.Leuven, 2006.

Dean Voets, Paolo Pilozzi, and Danny De Schreye. A new approach to termination analysis of Constraint Handling Rules. In Khalil Djelloul, Gregory J. Duck, and Martin Sulzmann, editors, *4th Workshop on Constraint Handling Rules*, pages 77–89. U.Porto, 2007.

Chapter 8

A Complete and Terminating Operational Semantics

Author:	Frank Raiser
Thesis Title:	Graph Transformation Systems in Constraint Handling Rules: Improved Methods for Program Analysis
School:	Ulm University, Germany
Publication Year:	2010

Foreword

This chapter is a part of the PhD thesis by Raiser [2010]. It summarizes recent research on simplifying the formal definitions of CHR's operational semantics. To this end, equivalence between CHR states is investigated and then used as a basis for defining CHR as a rewriting system over equivalence classes.

The resulting operational semantics can be defined via a single inference rule, due to the abstraction from state equivalence. As opposed to other definitions, the equivalence of states is a very intuitive subject such that one seldomly needs to make explicit use of the underlying equivalence axioms. Instead, the core task of analyzing a CHR program's behavior under its operational semantics becomes significantly simpler due to only having to consider a single inference rule rather than the traditional involved case distinction.

Later in this chapter, an alternative approach to the treatment of propagation rules is presented. It is based on persistent resources, roughly corresponding to linear logic. The resulting operational semantics, hence, provide an execution model for CHR that is much closer to logic than the approach based on a propagation history. Its practicability is shown through an implementation based on ω_p.

Finally, the simplified view on CHR as a rewriting system of equivalence classes is investigated further. This leads to a number of tools suitable for the formal analysis of CHR programs, which have been applied to great success in [Raiser, 2010].

The results given in this chapter stem from a recent dissertation, and hence, have not yet been superseded by other works. However, the simplified definition of CHR's operational semantics has already been presented and well-received at several international occassions, including the ICLP 2010 conference and the 2010 CHR summer school.

The chapter further assumes familiarity with the traditional operational semantics of CHR. In particular, the theoretical operational semantics ω_t is explained in Section 1.1.5 and the very abstract operational semantic ω_{va} is given in [Frühwirth, 2009]. The latter

is a simplified formulation of ω_t, and hence, the chapter can also be followed without consulting [Frühwirth, 2009].

8.1 Introduction

While CHR is known as a language that combines efficiency with declarativity, publications in the field display a tendency to favor one of these aspects over the other. We observe a spectrum of research directions ranging from the *analytical* to the *pragmatic*.

On the analytical end of the spectrum, emphasis is put on CHR as a mathematical formalism, declarativity, and the understanding of its logical foundations and theoretical properties. Several formalizations of the operational semantics, found in [Frühwirth, 1998] and [Frühwirth and Abdennadher, 2003], belong to this side of the spectrum.

On the downside, these operational semantics are detached from practical implementation in that they are oblivious to questions of efficiency and termination. Particularly, the class of rules called *propagation rules* causes trivial non-termination in both of them. Hence, it is safe to say that the existing analytical formalizations of the operational semantics lack a terminating execution model.

Example 8.1.1 (Lack of Termination with ω_{va})**.** *Consider the following propagation rule, which is part of a lower-equal solver (cf. Example 1.1.1) and corresponds to the transitivity relation*

$$\mathrm{leq}(A,B), \mathrm{leq}(B,C) \Longrightarrow \mathrm{leq}(A,C)$$

The main idea behind this rule is that in some cases creating an additional leq*-constraint may help to further simplify the store. However, under the very abstract operational semantics ω_{va}, we witness the following infinite derivation.*

$$\begin{array}{ll} & \langle \mathrm{leq}(A,B) \wedge \mathrm{leq}(B,C) \rangle \\ \rightarrowtail_{va} & \langle \mathrm{leq}(A,B) \wedge \mathrm{leq}(B,C) \wedge \mathrm{leq}(A,C) \rangle \\ \rightarrowtail_{va} & \langle \mathrm{leq}(A,B) \wedge \mathrm{leq}(B,C) \wedge \mathrm{leq}\, A,C) \wedge \mathrm{leq}(A,C) \rangle \\ \rightarrowtail^*_{va} & \langle \mathrm{leq}(A,B) \wedge \mathrm{leq}(B,C) \wedge \mathrm{leq}(A,C) \wedge \mathrm{leq}(A,C) \wedge \mathrm{leq}(A,C) \wedge \ldots \rangle \\ \rightarrowtail^*_{va} & \ldots \end{array}$$

This contrasts with most work on the pragmatic side of the spectrum, which emphasizes practical implementation and efficiency over formal reasoning. It originates with Abdennadher [1997], who proposed a token-based approach in order to avoid trivial non-termination: Every propagation rule is applicable only once to a specific combination of constraints. Thus, a terminating execution model for the full segment of CHR is provided, which however, lacks completeness.

Example 8.1.2 (Lack of Completeness with ω_t)**.** *Consider the following two rules and their execution under the theoretical operational semantics ω_t.*

$$\begin{array}{lcl} r_1 \ @ \ a & \Longrightarrow & b \\ r_2 \ @ \ b, b & \Leftrightarrow & c \end{array}$$

We know that in the abstract definition of CHR it should be possible with these rules to derive a c-constraint, when starting with a single a-constraint. Clearly, there exists a derivation under ω_{va} to achieve this, however, we cannot ever reach a c-constraint under

ω_t, *as the following derivation shows.*

$$\begin{array}{ll} & \langle a; \emptyset; \top; \emptyset \rangle_0^\emptyset \\ \rightarrowtail_t & \langle \emptyset; a\#0; \top; \emptyset \rangle_1^\emptyset \\ \rightarrowtail_t^{r_1} & \langle b; a\#0; \top; (r_1, 0) \rangle_1^\emptyset \\ \rightarrowtail_t & \langle \emptyset; a\#0, b\#1; \top; (r_1, 0) \rangle_2^\emptyset \\ \not\rightarrowtail_t & \end{array}$$

The final state demonstrates the essence of the effect of propagation tokens: Termination of rule r_1 for the identified constraint $a\#0$ is ensured at the cost of completeness.

In this chapter, we present the development of the operational semantics $\omega_!$, which provides a complete and terminating execution model for CHR. We begin in Section 8.2 with an investigation of equivalence of CHR states. It turns out, that an axiomatic formulation of state equivalence tremendously simplifies the formulation of analytical operational semantics of CHR. In that section, we hence introduce ω_e, which is defined as a rewriting system of equivalence classes and is compatible with ω_{va} [Frühwirth, 2009].

Next, Section 8.3 introduces so-called *persistent constraints*. They enable us to extend ω_e into the operational semantics $\omega_!$, which provides a complete and terminating execution model.

Building on the equivalence-based formulations of the operational semantics, we present a merge operator in Section 8.4, which is a useful tool in program analysis. Then, in Section 8.5, we discuss the differences between $\omega_!$ and existing operational semantics and present an implementation of $\omega_!$ via a source-to-source transformation. Finally, we discuss related work and future research paths in Section 8.6.

8.2 Equivalence-based Operational Semantics

While equivalence of states is an elementary concept in Constraint Handling Rules (CHR), the community has never agreed on a standard definition of that concept up to now. A plethora of definitions of state equivalence has been introduced in various areas of application (cf. Section 8.2.1). For example, the operational equivalence algorithm [Abdennadher and Frühwirth, 1999] compares two final states of different programs for equivalence. State equivalence is the basis for invariants such as in [Raiser and Frühwirth, 2009]. Several definitions have been introduced in the context of confluence considerations.

As the various authors had different intentions, the resulting definitions of state equivalence vary considerably. There is a general agreement that, from an operational point of view, any notion of state equivalence should be compliant with rule applications, i.e. for equivalent states the same rules are applicable and lead to equivalent results. However, this property has never been proven for any of the previously proposed definitions. Another general agreement is that from a declarative point of view the logical reading of equivalent states should also be equivalent.

Our aim is therefore to develop a definition of state equivalence that satisfies both the operational and the declarative view. Instead of defining a notion of state equivalence for a fixed problem setting, we intend a notion for which these generally agreed-upon properties hold. By construction, our definition of state equivalence then is compliant with rule application and the logical reading of states. Thus, it becomes a generic proof technique that can be applied to specific problems with the additional knowledge that the above-mentioned properties are satisfied.

In Section 8.2.1, we investigate equivalence of CHR states and provide an axiomatic definition, as well as a decidable criterion, for it. Then, Section 8.2.2 presents the operational semantics ω_e based on rewriting of equivalence classes.

8.2.1 State Equivalence

In this section, we first justify a set of desirable properties for a general notion of state equivalence in Section 8.2.1 and present them in the form of example cases. Then, we give a concise overview of the existing definitions of state equivalence and compare their behavior with respect to these example cases. We show that none of the existing definitions satisfies all of the example cases.

Next, Section 8.2.1 introduces an axiomatic definition of state equivalence along with several useful properties following from that definition. We show that it satisfies all the previously investigated example cases. Finally, we present a necessary, sufficient, and decidable criterion for determining equivalence of states in Section 8.2.1.

Existing State Equivalence Definitions

In this section, we evaluate existing definitions of state equivalence and postulate desirable properties of an equivalence relation over CHR states. To this end, we present several prototypical example cases of equivalent and non-equivalent CHR states. We concisely introduce the different notions of state equivalence that have been proposed so far, before we investigate how these notions apply to our example states.

In favor of a unified presentation, we will use the definition of ω_e states throughout this section. It clearly separates the three components that each have to be treated differently by state equivalence. We accordingly adapt existing definitions and results to this definition.

Definition 8.2.1 (ω_e State)**.** *An* ω_e state *is a tuple*

$$\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle.$$

- *The* goal $\mathbb{G}$ *is a multiset of CHR constraints.*
- *The* built-in constraint store $\mathbb{B}$ *is a conjunction of built-in constraints.*
- $\mathbb{V}$ *is a set of* global variables*.*

We use $\sigma, \sigma_0, \sigma_1, \ldots$ *to denote* ω_e *states and* Σ_e *to denote the set of all* ω_e *states. We denote the state* $\langle \emptyset; \top; \emptyset \rangle$ *as* $\sigma_\emptyset$*.*

We sometimes also use Σ to denote the set of all states, in cases where multiple operational semantics may apply. For example, many results of our proposed operational semantics presented in Section 8.3 equally apply to Σ_e.

Examples of Equivalences of CHR States Let us now consider the following examples of equivalent and non-equivalent states to highlight the differences between existing definitions of state equivalence. The relation $\equiv$ is used here as a generic equivalence relation with our desired properties. We will refer to our equivalence relations, defined later, as $\equiv_e$ and $\equiv_!$ instead.

$$\langle c(X); \top; \emptyset \rangle \equiv \langle c(Y); \top; \emptyset \rangle \quad (8.1)$$
$$\langle c(X); X = 0; \{X\} \rangle \equiv \langle c(0); X = 0; \{X\} \rangle \quad (8.2)$$
$$\langle \top; X \geq 0 \wedge X \leq 0 \wedge Y = 0; \{X\} \rangle \equiv \langle \top; X = 0; \{X\} \rangle \quad (8.3)$$
$$\langle c(0); \top; \{X\} \rangle \equiv \langle c(0); \top; \emptyset \rangle \quad (8.4)$$
$$\langle c(X); \top; \{X\} \rangle \not\equiv \langle c(Y); \top; \{Y\} \rangle \quad (8.5)$$

The equivalences (8.1)-(8.3) are motivated by the fact that the same rules are applicable to these states. Specifically, equivalence (8.1) covers renaming of local variables, equivalence (8.2) the substitution of variables with terms, and equivalence (8.3) built-in stores that are logically equivalent under the constraint theory $\mathcal{CT}$. Included in equivalence (8.3), are built-in constraints over strictly local variables [1] ($Y = 0$), which may be removed due to not affecting logical equivalence in any way ($\exists Y.Y = 0$ is a tautology).

As the states in equivalence (8.4) have the same logical reading $c(0)$, we require them to be equivalent. Unused global variables can practically occur, for example, when applying rule $c(X) \Leftrightarrow c(0)$ to the state $\langle c(X); \top; \{X\} \rangle$. Concerning non-equivalence (8.5), note that X, Y are free variables and therefore the logical readings $c(X)$ and $c(Y)$ are not equivalent.

Existing Definitions Over the last decade, the CHR community proposed various definitions for state equivalence. The following list identifies six distinct categories of equivalence definitions in the literature:

1. Definitions based on variable renaming [Abdennadher et al., 1999, Frühwirth et al., 2002, Duck, 2005, Meister, 2008] are often as simple as stating that two states are equivalent (or variants) if they can be obtained by variable renaming only. These definitions arose from the notion of variance on terms.

2. In [Raiser and Tacchella, 2007] a definition is given that is based on renaming of local variables as well as logical equivalence of built-in stores.

3. In [Haemmerlé and Fages, 2007] a similar definition is given for arbitrary binary relations rather than for CHR states only.

4. Duck et al. [2006] give another definition based on the refined operational semantics [Duck et al., 2004] of CHR.

5. Duck et al. [2007] – a follow-up to [Duck et al., 2006] – extends the definition with the usage of a unifier instead of variable renaming.

6. In [Abdennadher et al., 1999, Abdennadher, 2001] a *normalization function* is defined. While we emphasize that this definition was not targeted towards determining state equivalence, we include it in this work due to its clear structure that is similar to our proposed definition. When we talk about equivalence with respect to normalization we implicitly assume that two states are equivalent if and only if their normalizations are syntactically equivalent.

[1] In a state $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ the variables in $\mathbb{B}$ that do not occur in $\mathbb{G}$ and $\mathbb{V}$ are referred to as strictly local.

	(8.1)	(8.2)	(8.3)	(8.4)	(8.5)
Def. Type 1	$\equiv$	$\not\equiv$	$\not\equiv$	$\not\equiv$	$\equiv$
Def. Type 2	$\equiv$	$\not\equiv$	$\equiv$	$\equiv$	$\not\equiv$
Def. Type 3	$\equiv$	$\not\equiv$	$\equiv$	$\equiv$	$\not\equiv$
Def. Type 4	$\equiv$	$\not\equiv$	$\equiv$	$\not\equiv$	$\not\equiv$
Def. Type 5	$\equiv$	$\equiv$	$\equiv$	$\not\equiv$	$\not\equiv$
Def. Type 6	$\not\equiv$	$\equiv$	$\equiv$	$\not\equiv$	$\not\equiv$
Desired	$\equiv$	$\equiv$	$\equiv$	$\equiv$	$\not\equiv$

Table 8.1: Comparison of Different State Equivalence Definitions

Comparison of Existing Equivalence Definitions We have applied each of the existing definitions to each of the example cases. The results are presented in Table 8.1. Each entry shows whether the two corresponding example states are considered equivalent or not, according to the definition type used in that row. The last row presents the results that we deem desirable.

Table 8.1 reveals that definition types 1 to 4 do not satisfy the desired result for equivalence 8.2, which is the important substitution of variables with terms. Definition type 6 also shows two differences to our desired properties. However, the definition of Duck et al. [2007] closely corresponds to the properties we would like to see in an equivalence relation. The only difference is in equivalence (8.4), which allows removal of unused global variables. In most applications this kind of equivalence may be unnecessary though, such that the definition given in [Duck et al., 2007] seems to be a viable candidate.

However, the formulation used for the equivalence relation in [Duck et al., 2007] is very involved. It includes a unifier that has to be determined and its application in proofs quickly becomes complicated by that. Furthermore, in a personal correspondence with the authors of [Duck et al., 2007], we were informed that the formulation given in their work was obtained through a trial and error method. This can also be seen from the fact that [Duck et al., 2006] used a slightly different, but faulty, formulation (cf. Table 8.1, Def. Type 4).

Therefore, none of the previously published definitions of state equivalence respects all of the example cases.

Excursion: Detailed Discussion of Comparison

In this excursion, we discuss in more detail, why the individual definitions do not meet our desired results.

The definitions of type 1 lack any means to handle equivalent representations of built-ins. This problem arises in its purest form in case (8.3). Furthermore, they usually neglect to consider global variables, hence, case (8.5) fails as well.

Definitions of type 2 extend the variable renaming by built-in equivalence, which allows them to correctly treat case (8.3). Straightforward consideration of global variables solves case (8.5) satisfactorily as well.

However, all definitions of types 1, 2, 3, and 4 fail to consider more complex interactions between built-in equivalence and variable renaming, such that they fail for case (8.2). In that case, we substitute a variable X for its value, if the built-ins imply that $X = c$ for some value.

In fact, we will later see that the condition of permitting variable renaming is too weak. Case (8.2) revealed, that it cannot handle substitution and we will show that substitution and built-in equivalence actually subsume variable renaming. This is also the reason, why in our axiomatic definition of equivalence (cf. Definition 8.2.2) we have no axiom for variable renaming.

This problem of supporting substitutions has been discovered by Duck et al. [2006] as well and they resolved it by adapting their definition in [Duck et al., 2007], which is the definition type 5. As explained above, the definition turned out to be very involved and it still fails on case (8.4) in that it considers unused global variables as important. Intuitively, it appears to make more sense though to only care about elements relevant to the two states when determining their equivalence.

Finally, the normalization function, as it is defined in [Abdennadher, 2001] and represented by definition type 6 in Table 8.1, fails to yield the same normal forms for a local variable, thus failing case (8.1).

Novel Axiomatic Definition of State Equivalence

In this section, we introduce our axiomatic definition of equivalence that satisfies all desirable properties we identified in the previous section.

Definition 8.2.2 (Equivalence of ω_e States)**.** *Equivalence between ω_e states is the smallest equivalence relation $\equiv_e$ over CHR states that satisfies the following conditions:*

1. (Equality as Substitution)

$$\langle \mathbb{G}; X = t \wedge \mathbb{B}; \mathbb{V} \rangle \equiv_e \langle \mathbb{G}\,[X/t]\,; X = t \wedge \mathbb{B}; \mathbb{V} \rangle$$

2. (Transformation of the Constraint Store) *If $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ where $\bar{s}, \bar{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$, respectively, then:*

$$\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \equiv_e \langle \mathbb{G}; \mathbb{B}'; \mathbb{V} \rangle$$

3. (Omission of Non-Occurring Global Variables) *If X is a variable that does not occur in $\mathbb{G}$ or $\mathbb{B}$ then:*

$$\langle \mathbb{G}; \mathbb{B}; \{X\} \cup \mathbb{V} \rangle \equiv_e \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$$

4. (Equivalence of Failed States)

$$\langle \mathbb{G}; \bot; \mathbb{V} \rangle \equiv_e \langle \mathbb{G}'; \bot; \mathbb{V}' \rangle$$

Names of local variables in CHR states are chosen arbitrarily upon execution. Hence, considering these names invariant with respect to state equivalence suggests itself. In combination, axiom 1 and axiom 2 guarantee this desired property (cf. Lemma 8.2.3:1).

Axiom 1 and axiom 2 are furthermore invariant with respect to rule applicability and comply with logical equivalence of the logical readings. The same holds for axiom 4: On the logical level, inconsistent logical readings are of course logically equivalent.

Axiom 3 suggests itself with regard to logical readings, since adding or removing global constraints results in syntactically identical, and therefore indistinguishable, logical readings. Operationally, unused global variables have no effect, so it stands to reason to consider them redundant.

Lemma 8.2.3 states several properties that follow from Definition 8.2.2.

Lemma 8.2.3 (Properties of State Equivalence)**.** *The equivalence relation $\equiv_e$ over CHR states given in Def. 8.2.2 has the following properties:*

1. (Renaming of Local Variables) *Let X, Y be variables such that $X, Y \notin \mathbb{V}$ and Y does not occur in $\mathbb{G}$ or $\mathbb{B}$:*

$$\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \equiv_e \langle \mathbb{G}\,[X/Y]\,; \mathbb{B}\,[X/Y]\,; \mathbb{V} \rangle$$

2. (Partial Substitution) *Let $\mathbb{G}\,[X \wr t]$ be a multiset where* some *occurrences of X are substituted with t:*

$$\langle \mathbb{G}; X = t \wedge \mathbb{B}; \mathbb{V} \rangle \equiv_e \langle \mathbb{G}\,[X \wr t]\,; X = t \wedge \mathbb{B}; \mathbb{V} \rangle$$

3. (Logical Equivalence) *If*

$$\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \equiv_e \langle \mathbb{G}'; \mathbb{B}'; \mathbb{V}' \rangle$$

then $\mathcal{CT} \models \exists \bar{y}.\mathbb{G} \wedge \mathbb{B} \leftrightarrow \exists \bar{y}'.\mathbb{G}' \wedge \mathbb{B}'$, where $\bar{y}, \bar{y}'$ are the local variables of $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$, and $\langle \mathbb{G}'; \mathbb{B}'; \mathbb{V}' \rangle$, respectively.

Proof.

Property 1: By transformation of the constraint store, we have that $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ is equivalent to $\langle \mathbb{G}; X = Y \wedge \mathbb{B}; \mathbb{V} \rangle$. We apply equality as substitution and get $\langle \mathbb{G}\,[X/Y]\,; X = Y \wedge \mathbb{B}; \mathbb{V} \rangle$ which by transformation is equivalent to $\langle \mathbb{G}\,[X/Y]\,; \mathbb{B}\,[X/Y]\,; \mathbb{V} \rangle$.

Property 2: By substitution, we have that both $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ and $\langle \mathbb{G}\,[X \wr t]\,; X = t \wedge \mathbb{B}; \mathbb{V} \rangle$ are equivalent to $\langle \mathbb{G}\,[X/t]\,; X = t \wedge \mathbb{B}; \mathbb{V} \rangle$, because $\equiv_e$ is symmetric and transitive.

Property 3: All conditions given in Def. 8.2.2 correspond to valid logical equivalences:

Definition 8.2.2:1 preserves logical equivalence since

$$\mathbb{G} \wedge X = t \leftrightarrow \mathbb{G}\,[X/t] \wedge X = t$$

Definition 8.2.2:2: As $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ and the variables in $\bar{s}, \bar{s}'$ do not occur in $\mathbb{G}, \mathbb{G}'$, we have

$$\mathcal{CT} \models \exists \bar{y}.\mathbb{B} \wedge \mathbb{G} \leftrightarrow \exists \bar{y}'.\mathbb{B}' \wedge \mathbb{G}$$

Definition 8.2.2:3: For a variable X that does not occur in $\mathbb{B}$ or $\mathbb{G}$ we obviously have

$$\mathcal{CT} \models \exists X.\exists \bar{y}.\mathbb{B} \wedge \mathbb{G} \leftrightarrow \exists \bar{y}.\mathbb{B} \wedge \mathbb{G}$$

Definition 8.2.2:4 preserves logical equivalence due to the *ex falso quodlibet* property.

As logical equivalence is reflexive, transitive, and symmetric, Prop. 3 holds.

□

Example 8.2.4. *In this example, we show that the state equivalence relation $\equiv_e$ corresponds to our desired $\equiv$ relation. We refer to the four axioms given in Definition 8.2.2 via $\equiv_e^1, \equiv_e^2, \equiv_e^3$, and $\equiv_e^4$.*

- *Equivalence 8.1:*
  $\langle c(X); \top; \emptyset\rangle \equiv_e^2 \langle c(X); X = Y; \emptyset\rangle \equiv_e^1 \langle c(Y); X = Y; \emptyset\rangle \equiv_e^2 \langle c(Y); \top; \emptyset\rangle$
- *Equivalence 8.2 follows directly from* $\equiv_e^1$.
- *Equivalence 8.3 follows directly from* $\equiv_e^2$.
- *Equivalence 8.4 follows directly from* $\equiv_e^3$.
- *Equivalence 8.5: Using an axiomatic definition for* $\equiv_e$ *makes it difficult to prove non-equivalence. While an equivalence proof requires a sequence of axiom applications, proving non-equivalence requires us to show that no such sequence exists. Therefore, we delay the proof that* $\equiv_e$ *satisfies the non-equivalence 8.5, until we have the decision criterion, developed in the following section, available.*

Deciding State Equivalence

The decision problem for state equivalence is: given two states σ_1 and σ_2 decide whether $\sigma_1 \equiv_e \sigma_2$. In the positive case of two equivalent states, this can be decided by finding suitable applications of the axioms of Definition 8.2.2 that transform one state into the other. However, given two non-equivalent states, an axiom-based proof would require showing that no such applications can exist. As this is hard to automate, we would like to have a better criterion for a decision algorithm.

Logical equivalence between $\exists \bar{y}.\mathbb{G} \wedge \mathbb{B}$ and $\exists \bar{y}'.\mathbb{G}' \wedge \mathbb{B}'$ is a necessary but not a sufficient condition for state equivalence between $\langle \mathbb{G}; \mathbb{B}; \mathbb{V}\rangle$ and $\langle \mathbb{G}'; \mathbb{B}'; \mathbb{V}'\rangle$ (cf. Lemma 8.2.3:3). This is due to the fact that unlike logical equivalence, state equivalence preserves the multiplicities of logically equivalent user-defined constraints. A similar condition, which is also sufficient can be formulated in *linear-logic* [Betz and Frühwirth, 2005].

Theorem 20 gives a necessary and sufficient criterion for deciding state equivalence. Due to its preconditions, it technically decides a smaller relation than $\equiv_e$, because it only applies to the case that local variables are renamed apart and the set of global variables is unchanged.

However, this restriction is not problematic for deciding equivalence in general. By Lemma 8.2.3 we are free to rename local variables apart and by Def. 8.2.2:3 we can adjust the sets of global variables to match. Therefore, Theorem 20 gives us a necessary and sufficient criterion for equivalence of arbitrary states: first we transform the states into equivalent states that satisfy the preconditions, then we apply the theorem. The transformation is straightforward and equivalence-preserving, hence, the result we get from the theorem applies to the original states by transitivity of $\equiv_e$. Finally, decidability of our criterion is a direct consequence of decidability of $\mathcal{CT}$.

Theorem 20 (Criterion for $\equiv_e$)**.** *Let* $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V}\rangle, \sigma' = \langle \mathbb{G}'; \mathbb{B}'; \mathbb{V}\rangle$ *be* ω_e *states with local variables* $\bar{y}, \bar{y}'$ *that have been renamed apart.*

$$\sigma \equiv_e \sigma'$$
if and only if
$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$$

Proof. Let $\mathcal{C}$ be a binary predicate on CHR states such that $\mathcal{C}(\langle \mathbb{G}; \mathbb{B}; \mathbb{V}\rangle, \langle \mathbb{G}'; \mathbb{B}'; \mathbb{V}\rangle)$ holds iff

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$$

⇒:

We show that each of the three implicit conditions – reflexivity, symmetry, and transitivity – as well as the four explicit conditions of Def. 8.2.2 are sound w.r.t. criterion $\mathcal{C}$.

Reflexivity: Reflexivity is given as the following judgment is clearly true:

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B})) \wedge \forall(\mathbb{B} \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B}))$$

Symmetry: Symmetry of $\mathcal{C}$ is obvious.

Transitivity: Assume three states $\sigma = \langle\mathbb{G};\mathbb{B};\mathbb{V}\rangle, \sigma' = \langle\mathbb{G}';\mathbb{B}';\mathbb{V}\rangle, \sigma'' = \langle\mathbb{G}'';\mathbb{B}'';\mathbb{V}\rangle$ with distinct local variables $\bar{y}, \bar{y}', \bar{y}''$ such that $\mathcal{C}(\sigma,\sigma')$ and $\mathcal{C}(\sigma',\sigma'')$. By definition, we have:

$$\begin{array}{ll}
\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists\bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B})) & (i)\\
\mathcal{CT} \models \forall(\mathbb{B}' \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B})) & (ii)\\
\mathcal{CT} \models \forall(\mathbb{B}' \rightarrow \exists\bar{y}''.((\mathbb{G}' = \mathbb{G}'') \wedge \mathbb{B}'')) & (iii)\\
\mathcal{CT} \models \forall(\mathbb{B}'' \rightarrow \exists\bar{y}'.((\mathbb{G}' = \mathbb{G}'') \wedge \mathbb{B}')) & (iv)
\end{array}$$

From (i) and (iii) follows:

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists\bar{y}''.((\mathbb{G} = \mathbb{G}'') \wedge \mathbb{B}''))$$

From (ii) and (iv) follows:

$$\mathcal{CT} \models \forall(\mathbb{B}'' \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}'') \wedge \mathbb{B}))$$

Consequently, $\mathcal{C}(\sigma,\sigma'')$ holds.

Equality as Substitution: Assume two states $\sigma = \langle\mathbb{G}; X = t \wedge \mathbb{B};\mathbb{V}\rangle, \sigma' = \langle\mathbb{G}\,[X/t]\,; X = t \wedge \mathbb{B};\mathbb{V}\rangle$ with local variables $\bar{y}, \bar{y}'$. As $\mathcal{CT} \models \forall(X = t \rightarrow (\mathbb{G} = \mathbb{G}\,[X/t]))$, we have $\mathcal{C}(\sigma,\sigma')$.

Transformation of the Constraint Store: Assume two states $\sigma = \langle\mathbb{G};\mathbb{B};\mathbb{V}\rangle$ and $\sigma' = \langle\mathbb{G};\mathbb{B}';\mathbb{V}\rangle$ with local variables $\bar{y}, \bar{y}'$ and strictly local variables $\bar{s}, \bar{s}'$ such that $\mathcal{CT} \models \exists\bar{s}.\mathbb{B} \leftrightarrow \exists\bar{s}'.\mathbb{B}'$. This implies the following judgment:

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists\bar{y}'.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B}))$$

Hence, $\mathcal{C}(\sigma,\sigma')$ holds.

Omission of Non-Occurring Global Variables: Does not apply since σ and σ' share the set $\mathbb{V}$ of global variables.

Equivalence of Failed States: For any two failed states, they are of the form $\langle\mathbb{G};\bot;\mathbb{V}\rangle$ and $\langle\mathbb{G}';\bot;\mathbb{V}\rangle$. The following judgment proves $\mathcal{C}(\langle\mathbb{G};\bot;\mathbb{V}\rangle, \langle\mathbb{G}';\bot;\mathbb{V}\rangle)$:

$$\mathcal{CT} \models \forall(\bot \rightarrow \exists\bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \bot)) \wedge \forall(\bot \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \bot))$$

⇐:

We consider two CHR states $\sigma = \langle\mathbb{G};\mathbb{B};\mathbb{V}\rangle, \sigma' = \langle\mathbb{G}';\mathbb{B}';\mathbb{V}\rangle$ with disjunct local variables $\bar{y}$ and $\bar{y}'$. We assume that

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists\bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$$

If there does not exist a pairwise matching $\mathbb{G} = \mathbb{G}'$, we have $\mathbb{B} = \mathbb{B}' = \bot$, which proves that $\sigma \equiv_e \sigma'$ by Def. 8.2.2:4. In the following, we assume that a pairwise matching $\mathbb{G} = \mathbb{G}'$ does exist.

It follows from $\forall(\mathbb{B} \rightarrow \exists\bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')$ by Def. 8.2.2:2 that:

$$\sigma \equiv_e \langle \mathbb{G}; \mathbb{G} = \mathbb{G}' \wedge \mathbb{B} \wedge \mathbb{B}'; \mathbb{V} \rangle$$

By Def. 8.2.2:1 we have:

$$\sigma \equiv_e \langle \mathbb{G}'; \mathbb{G} = \mathbb{G}' \wedge \mathbb{B} \wedge \mathbb{B}'; \mathbb{V} \rangle$$

From $\forall(\mathbb{B}' \rightarrow \exists\bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$ we get by Def. 8.2.2:2 that:

$$\sigma \equiv_e \langle \mathbb{G}'; \mathbb{B}'; \mathbb{V} \rangle = \sigma'$$

□

Example 8.2.5. *We can now use Theorem 20 to prove our desired non-equivalence* (8.5)*. Let* $\sigma = \langle c(X); \top; \{X\} \rangle$ *and* $\sigma' = \langle c(Y); \top; \{Y\} \rangle$*, then we first consider two states with the same global variables as follows:*

$$\sigma_1 ::= \langle c(X); \top; \{X, Y\} \rangle \equiv_e \sigma$$

$$\sigma_2 ::= \langle c(Y); \top; \{X, Y\} \rangle \equiv_e \sigma'$$

According to the above theorem, $\sigma_1 \equiv_e \sigma_2$ *holds if and only if*

$$\mathcal{CT} \models \forall X, Y.\top \rightarrow (c(X) = c(Y)) \wedge \forall X, Y.\top \rightarrow (c(X) = c(Y)).$$

However, it clearly holds that $\mathcal{CT} \not\models \forall X, Y.c(X) = c(Y)$*, such that Theorem 20 proves* $\sigma_1 \not\equiv_e \sigma_2$*, and because* $\equiv_e$ *is an equivalence relation, this proves* $\sigma \not\equiv_e \sigma'$*.*

Intuitively, the theorem tries to find out if the two multisets of CHR constraints can be made syntactically equivalent ($\mathbb{G} = \mathbb{G}'$). However, we may only consider existential quantification for local variables, which subsumes variable renaming and substitution. Interpreting $\mathbb{G} = \mathbb{G}'$ as a unification, we must additionally ensure that the required bindings satisfy the other state's built-in store.

Theorem 20 is indeed suitable for implementation. Such an implementation has been given in [Langbein et al., 2010], where it was used as the foundation of a confluence checker.

8.2.2 State Transition System

In this section, we first define an operational semantics for CHR based on our axiomatic definition of state equivalence. We then discuss its equivalence to the traditional definition, hence showing that state equivalence is indeed compliant with rule application. Finally, we introduce a novel view of the CHR transition system: Given the compliance of state equivalence with rule application, we can integrate this knowledge into the formulation of the operational semantics, hence, interpreting CHR as a rewriting system of equivalence classes.

$$\frac{r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b}{\langle H_1 \uplus H_2 \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V}\rangle \rightarrowtail_e^r \langle H_1 \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V}\rangle}$$

$$\frac{\sigma' \equiv \sigma \qquad \sigma \rightarrowtail^r \tau \qquad \tau \equiv \tau'}{\sigma' \rightarrowtail_e^r \tau'}$$

Table 8.2: State Transition System ω_e

Equivalence-based Transition System

In this section, we present a formulation of the operational semantics based on state equivalence. Our definition is not only based on the traditional definition, but is also provably equivalent. Integrating the notion of state equivalence permits removing the matching that has traditionally been hidden in the complex formula $H_1 = H_1' \wedge H_2 = H_2'$. Furthermore, imposing a guard condition on $\mathcal{CT}$ becomes dispensable, leading to the following simplified operational semantics:

Definition 8.2.6 (Operational Semantics ω_e)**.** *For a CHR program $\mathcal{P}$ we define the state transition system $(\Sigma_e, \rightarrowtail_e)$, referred to as ω_e , as given in Table 8.2. The transition is based on a variant of a rule r in $\mathcal{P}$ such that its local variables are disjoint from the variables occurring in the pre-transition state.*

When the rule r is clear from the context or not important, we may write $\rightarrowtail_e$ rather than $\rightarrowtail_e^r$. By $\rightarrowtail_e^*$, we denote the reflexive-transitive closure of $\rightarrowtail_e$.

Example 8.2.7. *Consider the example for computing the greatest common divisor, which consists of the following two rules.*

$$\begin{array}{lcl} \mathrm{gcd}_1 \ @ \ \mathrm{gcd}(0) & \Leftrightarrow & \top \\ \mathrm{gcd}_2 \ @ \ \mathrm{gcd}(N) \setminus \mathrm{gcd}(M) & \Leftrightarrow & M \geq N \wedge N > 0 \mid \mathrm{gcd}(L), L = M\%N \end{array}$$

Given the transition system $\rightarrowtail_e$, we can now derive the greatest common divisor of 6 *and* 15 *as follows:*

$$\begin{array}{ll} & \langle \mathrm{gcd}(6), \mathrm{gcd}(15); \top; \emptyset\rangle \\ \equiv_e & \langle \mathrm{gcd}(N), \mathrm{gcd}(M); M \geq N \wedge N > 0 \wedge N = 6 \wedge M = 15; \emptyset\rangle \\ \rightarrowtail_e^{\mathrm{gcd}_2} & \langle \mathrm{gcd}(N), \mathrm{gcd}(L); M \geq N \wedge N > 0 \wedge L = M\%N \wedge N = 6 \wedge M = 15; \emptyset\rangle \\ \equiv_e & \langle \mathrm{gcd}(6), \mathrm{gcd}(3); \top; \emptyset\rangle \\ \equiv_e & \langle \mathrm{gcd}(N), \mathrm{gcd}(M); M \geq N \wedge N > 0 \wedge N = 3 \wedge M = 6; \emptyset\rangle \\ \rightarrowtail_e^{\mathrm{gcd}_2} & \langle \mathrm{gcd}(N), \mathrm{gcd}(L); M \geq N \wedge N > 0 \wedge L = M\%N \wedge N = 3 \wedge M = 6; \emptyset\rangle \\ \equiv_e & \langle \mathrm{gcd}(3), \mathrm{gcd}(0); \top; \emptyset\rangle \\ \rightarrowtail_e^{\mathrm{gcd}_1} & \langle \mathrm{gcd}(3); \top; \emptyset\rangle \end{array}$$

We can see from the above derivation that there were three rule applications. We reuse the variables N, M in the derivation above, because they can be removed from the state after the rule application, by applying the equivalence relation.

The most significant difference of this formulation of the operational semantics to previous ones is that the rule head is required to exist within the state exactly as is. Traditionally, we require constraints in the state that are *similar* to the ones in the rule

head. They are then matched via a substitution. With Definition 8.2.6 however, this is outsourced into the equivalence relation. We still have to prove that the current state contains the required rule head, but the formulation of the operational semantics itself is simplified.

Another advantage can be seen in the previous example: during a derivation we can use the equivalence relation to freely switch to different representations of a state. This, for example, allows us to constantly simplify the built-in store. When we compare this to ω_{va}, ω_t, and ω_p, we find that in ω_{va} there is no formal means to ever simplify a built-in store again, while ω_t and ω_p may only simplify the built-in store during a **Solve** transition.

Example 8.2.8. *To demonstrate the tremendous simplification gained by ω_e let us consider the previous derivation for computing the greatest common divisor of 6 and 15 for ω_t. For space reason, we denote $L_1 = 3 \wedge N_2 = L_1 \wedge M_2 = 6 \wedge M_2 \geq N_2 \wedge N_2 > 0$ as $\mathbb{B}$ in the below derivation.*

$$
\begin{array}{ll}
 & \langle \gcd(6), \gcd(15); \emptyset; \top; \emptyset \rangle_0^\emptyset \\
\rightarrowtail_t^{\textit{Introduce}} & \langle \gcd(15); \gcd(6)\#0; \top; \emptyset \rangle_1^\emptyset \\
\rightarrowtail_t^{\textit{Introduce}} & \langle \emptyset; \gcd(6)\#0, \gcd(15)\#1; \top; \emptyset \rangle_2^\emptyset \\
\rightarrowtail_t^{\gcd_2} & \langle L_1 = M_1 \% N_1, \gcd(L_1); \gcd(6)\#0; N_1 = 6 \wedge M_1 = 15 \wedge M_1 \geq N_1 \wedge N_1 > 0); (\gcd_2, [0,1]) \rangle_2^\emptyset \\
\rightarrowtail_t^{\textit{Solve}} & \langle \gcd(L_1); \gcd(6)\#0; L_1 = 3; (\gcd_2, [0,1]) \rangle_2^\emptyset \\
\rightarrowtail_t^{\textit{Introduce}} & \langle \emptyset; \gcd(6)\#0, \gcd(L_1)\#2; L_1 = 3; (\gcd_2, [0,1]) \rangle_3^\emptyset \\
\rightarrowtail_t^{\gcd_2} & \langle L_2 = M_2 \% N_2, \gcd(L_2); \gcd(L_1)\#2; \mathbb{B}; (\gcd_2, [0,1]), (\gcd_2, [2,0]) \rangle_3^\emptyset \\
\rightarrowtail_t^{\textit{Solve}} & \langle \gcd(L_2); \gcd(L_1)\#2; L_1 = 3 \wedge L_2 = 0; (\gcd_2, [0,1]), (\gcd_2, [2,0]) \rangle_3^\emptyset \\
\rightarrowtail_t^{\textit{Introduce}} & \langle \emptyset); \gcd(L_1)\#2, \gcd(L_2)\#3; L_1 = 3 \wedge L_2 = 0; (\gcd_2, [0,1]), (\gcd_2, [2,0]) \rangle_4^\emptyset \\
\rightarrowtail_t^{\gcd_1} & \langle \emptyset; \gcd(L_1)\#2; L_1 = 3 \wedge L_2 = 0; (\gcd_2, [0,1]), (\gcd_2, [2,0]), (\gcd_1, [3]) \rangle_4^\emptyset
\end{array}
$$

Apart from the higher complexity of this derivation, there is an important point about the resulting final state: For the above, one often reads abbreviated derivations like

$$\langle \gcd(6), \gcd(15); \emptyset; \top; \emptyset \rangle_0^\emptyset \rightarrowtail_t^* \langle \emptyset; \gcd(3)\#i; \mathbb{B}; \mathbb{T} \rangle_n^\emptyset,$$

for some $i, \mathbb{B}, \mathbb{T}$, and n. However, the above statement is in fact wrong, because there exist no such derivations according to Definition 1.1.5. Instead there only exists a constraint $\gcd(L_1)\#3$ and the built-in $L_1 = 3$. With ω_t we are not given any means to change this into a constraint of the form $\gcd(3)$.

*The same problem exists with ω_{va}, in that we also cannot get a $\gcd(3)$ constraint in the resulting final state. In fact, the situation is worse for ω_{va}, as there is no built-in simplification possible at all, whereas ω_t at least has the **Solve** transition available.*

Compliance to Rule Application

Definition 8.2.6 implicitly assumes that when we can apply a rule to a state, we can also apply it to all equivalent states. Furthermore, the resulting states of the rule application are then assumed to be equivalent as well. This is by no means a trivial assumption. While it is a property that we would intuitively expect from a proper equivalence relation, it has not been proven before the introduction of ω_e in [Raiser et al., 2009].

The following theorem from [Raiser et al., 2009] proves the equivalence of ω_e and ω_{va}. This implicitly proves the above desired property. Hence, it effectively serves as justification for the viability of Definition 8.2.6.

Theorem 21 (Equivalence of the Definitions)**.** *For a CHR state σ we have*

1. *If $\sigma \rightarrowtail_e^r \tau$ then there exists a state $\tau' \equiv \tau$ with $\sigma \rightarrowtail_{va}^r \tau'$*
2. *If $\sigma \rightarrowtail_{va}^r \tau'$ then there exists a state $\tau \equiv \tau'$ with $\sigma \rightarrowtail_e^r \tau$*

Proof. see [Raiser et al., 2009]. □

$$\frac{r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b}{[\langle H_1 \uplus H_2 \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V}\rangle] \rightarrowtail_e^r [\langle H_1 \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V}\rangle]}$$

Table 8.3: State Transition System ω_e with Equivalence Classes

Rewriting of Equivalence Classes

By now, we have introduced an axiomatic definition of state equivalence and shown its compliance with rule applications. Definition 8.2.6 of ω_e can therefore be abstracted further. The second inference rule allows us to freely switch between equivalent states during a derivation. Therefore, the actual syntactical representation of a state is of no importance anymore, which leads to a different view on the transition system: instead of considering all syntactical representations as different states, we propose the following reformulation based on equivalence classes of states. In this work, $[\sigma]$ denotes the equivalence class of a state $\sigma \in \Sigma_e$, i.e. $[\sigma] ::= \{\sigma' \in \Sigma_e \mid \sigma' \equiv_e \sigma\}$. When rewriting such equivalence classes, we consider an equivalence class as a state of our state transition system. Hence, the set of all states becomes $\Sigma_e/\equiv_e$.

Definition 8.2.9 (Operational Semantics ω_e)**.** *For a CHR program $\mathcal{P}$, the state transition system $(\Sigma_e/\equiv_e, \rightarrowtail_e)$ is given in Table 8.3. The transition is based on a variant of a rule r in $\mathcal{P}$ such that its local variables are disjoint from the variables occurring in the representant of the pre-transition state.*

We consider both, Definition 8.2.6 and Definition 8.2.9, as the operational semantics ω_e. They clearly are equivalent and the brackets used to denote equivalence classes ensure there is no disambiguity. However, as Definition 8.2.9 abstracts over equivalence, the remainder of this work will mostly refer to it, when discussing ω_e.

Example 8.2.10. *Reconsider the previous example for computing the greatest common divisor of* 6 *and* 15*. Using Definition 8.2.9, we can present the derivation in the following concise way:*

$$\begin{aligned} & [\langle \gcd(6), \gcd(15); \top; \emptyset\rangle] \\ \rightarrowtail_e^{\gcd_2} \quad & [\langle \gcd(6), \gcd(3); \top; \emptyset\rangle] \\ \rightarrowtail_e^{\gcd_2} \quad & [\langle \gcd(3), \gcd(0); \top; \emptyset\rangle] \\ \rightarrowtail_e^{\gcd_1} \quad & [\langle \gcd(3); \top; \emptyset\rangle] \end{aligned}$$

In [Raiser, 2010] it is demonstrated, that the equivalence-based operational semantics ω_e facilitates formal proofs. It allows us to switch to the most convenient representation of a CHR store at any time and to neglect other equivalent states, because it often suffices to show a property for one state in order to proof it for all equivalent states.

8.3 Constraint Handling Rules with Persistent Constraints

Recent work on linear-logical algorithms [Simmons and Pfenning, 2008] and the close relation of CHR to linear-logic [Betz and Frühwirth, 2005] suggest a novel approach that emphasizes aspects from both sides of the spectrum to a useful degree: In [Betz et al., 2009], we introduce the notion of *persistent constraints* to CHR, a concept reminiscent of unrestricted or "banged" propositions in linear-logic. Persistent constraints provide a finite representation of the result of any number of propagation rule firings.

We furthermore introduce a state transition system based on persistent constraints, which is explicitly irreflexive. In combination, the two ideas solve the problem of trivial non-termination while retaining declarativity and preserving the potential for effective concurrent execution. This state transition system requires no more than two rules. As every transition step corresponds to a CHR rule application, it facilitates formal reasoning over programs.

In this work, we show that the resulting operational semantics $\omega_!$ is sound and complete with respect to ω_e. We show that $\omega_!$ can be faithfully embedded into the operational semantics ω_p, thus effectively providing an implementation in the form of a source-to-source transformation. All operational semantics developed with an emphasis on pragmatic aspects lack this completeness property. Therefore, it is possible to implement CHR soundly and completely with respect to its abstract foundations, whilst featuring a terminating execution model for propagation rules.

Example 8.3.1. *Consider the following straightforward CHR program for computing the transitive hull of a graph represented by edge constraints $e/2$:*

$$t \quad @ \quad e(X,Y), e(Y,Z) \quad \Longrightarrow \quad e(X,Z)$$

This most intuitive formulation of a transitive hull is not a suitable implementation in most existing operational semantics. In fact, for goals containing cyclic graphs it is non-terminating in all aforementioned existing semantics. In this work we show that execution in our proposed semantics $\omega_!$ correctly computes the transitive hull whilst guaranteeing termination.

8.3.1 State Equivalence

In this section, we present the operational semantics $\omega_!$ with persistent constraints, originally proposed in [Betz et al., 2009]. It is based on the following ideas:

1. In ω_e, the body of a propagation rule can be generated any number of times, provided that the corresponding head constraints are present in the store. In order to give consideration to this theoretical behavior, we introduce those body constraints as so-called *persistent constraints*. A persistent constraint is a finite representation of a large, though unspecified number of identical constraints. For a proper distinction, constraints that are not persistent constraints are henceforth called *linear* constraints.

2. As a secondary consequence, arbitrary generation of rule bodies in ω_e affects other types of CHR rules as well. Consider the following program:

$$\begin{array}{llll} \text{r1} & @ & a & \Longrightarrow \quad b \\ \text{r2} & @ & b & \Leftrightarrow \quad c \end{array}$$

   When executed with goal a, this program can generate an arbitrary number of b-constraints. As a consequence of this, it can also generate arbitrarily many c-constraints. To take these indirect consequences of propagation rules into account, we introduce a rule's body constraints as persistent, whenever its removed head can be matched completely with persistent constraints.

3. As a persistent constraint represents an arbitrary number of identical constraints, we consider multiple occurrences of a persistent constraint as idempotent. Thus, we implicitly apply a set semantics to persistent constraints.

4. We adapt the execution model such that a transition takes place only if the post-transition state is not equivalent to the pre-transition state. This entails two beneficial consequences: Firstly, in combination with the set semantics on persistent constraints, it avoids trivial non-termination of propagation rules. Secondly, as failed states are equivalent, it enforces termination upon failure.

We adapt the definition of $\omega_!$ states with respect to ω_e. The goal store $\mathbb{G}$ of ω_e states is split into a store $\mathbb{L}$ of linear constraints and a store $\mathbb{P}$ of persistent constraints.

Definition 8.3.2 ($\omega_!$ State)**.** *A $\omega_!$ state is a tuple of the form $\langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle$, where $\mathbb{L}$ and $\mathbb{P}$ are multisets of CHR constraints called the* linear (CHR) store *and* persistent (CHR) store*, respectively. $\mathbb{B}$ is a conjunction of built-in constraints and $\mathbb{V}$ is a set of variables called the* global variables*. We use $\Sigma_!$ to denote the set of all $\omega_!$ states.*

The following definition of state equivalence is adapted to comply with Definition 8.3.2 and to handle idempotence of persistent constraints.

Definition 8.3.3 (Equivalence of $\omega_!$ States)**.** *Equivalence between $\omega_!$ states is the smallest equivalence relation $\equiv_!$ over $\omega_!$ states that satisfies the following conditions:*

1. (Equality as Substitution)

$$\langle \mathbb{L}; \mathbb{P}; X = t \wedge \mathbb{B}; \mathbb{V} \rangle \equiv_! \langle \mathbb{L}\,[X/t]\,; \mathbb{P}\,[X/t]\,; X = t \wedge \mathbb{B}; \mathbb{V} \rangle$$

2. (Transformation of the Constraint Store) *If $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ where $\bar{s}, \bar{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$, respectively, then:*

$$\langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle \equiv_! \langle \mathbb{L}; \mathbb{P}; \mathbb{B}'; \mathbb{V} \rangle$$

3. (Omission of Non-Occurring Global Variables) *If X is a variable that does not occur in $\mathbb{L}$, $\mathbb{P}$, or $\mathbb{B}$ then:*

$$\langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \{X\} \cup \mathbb{V} \rangle \equiv_! \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle$$

4. (Equivalence of Failed States)

$$\langle \mathbb{L}; \mathbb{P}; \bot; \mathbb{V} \rangle \equiv_! \langle \mathbb{L}'; \mathbb{P}'; \bot; \mathbb{V}' \rangle$$

5. (Contraction)

$$\langle \mathbb{L}; P \uplus P \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle \equiv_! \langle \mathbb{L}; P \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle$$

The following definition presents an auxiliary concept that we use to formulate a criterion for $\omega_!$ equivalence, in analogy to Theorem 20. Intuitively, $\mathbb{G} \bowtie \mathbb{G}'$ holds, if after converting $\mathbb{G}$ and $\mathbb{G}'$ to sets by eliminating duplicates, the two sets are identical.

Definition 8.3.4 ($\bowtie$)**.** *The relation $\bowtie$ over multisets of constraints is defined as*

$$\mathbb{G} \bowtie \mathbb{G}' \text{ if and only if } (\forall c \in \mathbb{G}.\exists c' \in \mathbb{G}'.c = c') \wedge (\forall c' \in \mathbb{G}'.\exists c \in \mathbb{G}.c = c')$$

Theorem 22 (Criterion for $\equiv_!$)**.** *Let $\sigma = \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle$, $\sigma' = \langle \mathbb{L}'; \mathbb{P}'; \mathbb{B}'; \mathbb{V} \rangle$ be $\omega_!$ states with local variables $\bar{y}, \bar{y}'$ that have been renamed apart.*

$$\sigma \equiv_! \sigma'$$
$$\text{if and only if}$$
$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}')) \;\wedge\; \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}))$$

Proof.

'⇐': We consider two $\omega_!$ states $\sigma = \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle, \sigma' = \langle \mathbb{L}'; \mathbb{P}'; \mathbb{B}'; \mathbb{V} \rangle$ with local variables $\bar{y}$ and $\bar{y}'$. We furthermore assume that:

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}')) \ \wedge \ \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}))$$

If $\mathcal{CT} \models \neg\exists((\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}'))$, we have $\mathcal{CT} \models \mathbb{B} = \mathbb{B}' = \bot$ such that Def. 8.3.3:4 proves $\sigma \equiv_! \sigma'$. In the following, we assume that a matching $(\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}')$ exists.

It follows from $\forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}'))$ by Def. 8.3.3:2 that:

$$\sigma \equiv_! \langle \mathbb{L}; \mathbb{P}; (\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B} \wedge \mathbb{B}'; \mathbb{V} \rangle$$

Def. 8.3.3:1 gives us:

$$\sigma \equiv_! \langle \mathbb{L}'; \mathbb{P}''; (\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B} \wedge \mathbb{B}'; \mathbb{V} \rangle$$

where $\mathbb{P}''$ equals $\mathbb{P}'$ modulo multiplicities. By Def. 8.3.3:5 we thus get:

$$\sigma \equiv_! \langle \mathbb{L}'; \mathbb{P}'; (\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B} \wedge \mathbb{B}'; \mathbb{V} \rangle$$

From $\forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{L} = \mathbb{L}') \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}))$ follows by Def. 8.3.3:2 that:

$$\sigma \equiv_! \langle \mathbb{L}'; \mathbb{P}'; \mathbb{B}'; \mathbb{V} \rangle = \sigma'$$

'⇒': To prove the forward direction, we have to show the compliance of the conditions in Def. 8.3.3:1 to Def. 8.3.3:5 with our criterion. For Def. 8.3.3:1 to Def. 8.3.3:4, compliance is analogous to Thm. 20. Hence, we now consider Def. 8.3.3:5:

Let $\sigma = \langle \mathbb{L}; P \uplus P \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle, \sigma' = \langle \mathbb{L}; P \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle \in \Sigma_!$ with local variables $\bar{y}, \bar{y}'$. As, $(P \uplus P \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P})$, the following is a tautology:

$$\mathcal{CT} \models \begin{array}{l} \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{L} = \mathbb{L}) \wedge ((P \uplus P \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P})) \wedge \mathbb{B})) \wedge \\ \forall(\mathbb{B} \rightarrow \exists \bar{y}.((\mathbb{L} = \mathbb{L}) \wedge ((P \uplus P \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P}')) \wedge \mathbb{B})) \end{array}$$

□

8.3.2 State Transition System

Based on the definition of ω_e, we define the operational semantics $\omega_!$ below. Since body constraints may be introduced either as linear or as persistent constraints, uniform rule application is replaced by two distinct application modes. An important restriction is that $\omega_!$ is only defined for *range-restricted* programs (cf. Section 8.5.2 for details). A range-restricted rule is one, whose body and guard only contain variables from the head and a CHR program is range-restricted, if all its rules are.

Definition 8.3.5 ($\omega_!$ Transitions)**.** *For a range-restricted CHR program $\mathcal{P}$, the state transition system $(\Sigma_!/\equiv_!, \rightarrowtail_!)$, referred to as $\omega_!$, is given in Table 8.4.*

When the rule r is clear from the context or not important, we may write $\rightarrowtail_!$ rather than $\rightarrowtail_!^r$. By $\rightarrowtail_!^*$, we denote the reflexive-transitive closure of $\rightarrowtail_!$.

ApplyLinear:

$$\frac{r \ @ \ (H_1^l \uplus H_1^p) \backslash (H_2^l \uplus H_2^p) \Leftrightarrow G \mid B_c, B_b \qquad H_2^l \neq \emptyset \qquad [\sigma] \neq [\tau]}{\begin{array}{c}[\sigma] = [\langle H_1^l \uplus H_2^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}\rangle] \\ \rightarrowtail_!^r \ [\langle H_1^l \uplus B_c \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}\rangle] = [\tau]\end{array}}$$

ApplyPersistent:

$$\frac{r \ @ \ (H_1^l \uplus H_1^p) \backslash H_2^p \Leftrightarrow G \mid B_c, B_b \qquad [\sigma] \neq [\tau]}{\begin{array}{c}[\sigma] = [\langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}\rangle] \\ \rightarrowtail_!^r \ [\langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus B_c \uplus \mathbb{P}; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}\rangle] = [\tau]\end{array}}$$

Table 8.4: State Transition System $\omega_!$

Example 8.3.6. *Again consider the transitive edge program from Example 8.3.1 and an initial state that contains a cyclic graph. For such an initial state the program is non-terminating for all previous operational semantics presented in this work. However, with $\omega_!$, we get the following terminating derivation.*

$$\begin{array}{ll}
[\sigma] = & [\langle e(A,B), e(B,A); \emptyset; \top; \{A,B\}\rangle] \\
\rightarrowtail_! & [\langle e(A,B), e(B,A); e(A,A); \top; \{A,B\}\rangle] \\
\rightarrowtail_! & [\langle e(A,B), e(B,A); e(A,A), e(B,B); \top; \{A,B\}\rangle] \\
\rightarrowtail_! & [\langle e(A,B), e(B,A); e(A,A), e(B,B), e(A,B); \top; \{A,B\}\rangle] \\
\rightarrowtail_! & [\langle e(A,B), e(B,A); e(A,A), e(B,B), e(A,B), e(B,A); \top; \{A,B\}\rangle] = [\tau]
\end{array}$$

*All four state transitions are made via the **ApplyPersistent** inference rule and add the resulting constraints to the persistent store. Irreflexivity of $(\Sigma_!/\equiv_!, \rightarrowtail_!)$ ensures that such a persistent constraint cannot be generated again. Therefore, after all four possible edges have been generated as persistent constraints, no further rule applications are possible.*

This operational semantics $\omega_!$ fills a gap left by the existing operational semantics: It is, on the one hand, *complete*, i.e. every computation that is possible under the operational semantics ω_e corresponds to one in $\omega_!$. On the other hand, it offers general *termination* for propagation rules, which ω_e lacks and caused the token-based operational semantics ω_t to be developed, which in turn lacks completeness.

This situation is easily demonstrated via a propagation rule $a \Rightarrow b$ and an initial state that contains an a-constraint. With ω_e it is possible to apply the rule any number of times. In particular, we get a non-terminating derivation which keeps generating b-constraints. In the more pragmatic operational semantics ω_t instead, we can only derive a single b-constraint. While we have termination in that case, we lack completeness in that we are unable to generate a second or third b-constraint.

The operational semantics $\omega_!$, however, fixes these problems with a best-of-both-worlds approach: in a single derivation step a persistent b-constraint is derived, hence, we have termination. Furthermore, this persistent constraint can be used in lieu of any number of linear b-constraints. If we add to the above rule, the rule $a, b, b \Leftrightarrow c$, then $\omega_!$ allows a second derivation that yields a linear c-constraint. Again, ω_t fails due to its lack of completeness to ever derive a c-constraint, and ω_e still allows the non-terminating derivation, but also terminating derivations that yield a c-constraint.

Of course, $\omega_!$ is also *sound* with respect to ω_e, i.e. all derivations made in $\omega_!$ correspond to derivations in ω_e. For a formal treatment of the soundness and completeness of $\omega_!$ with respect to ω_e we refer the reader to [Betz et al., 2010].

Excursion: Proverbial Example

This excursion demonstrates the potential of persistent constraints in a less serious way. Consider the following well-known Chinese proverb, which we want to model as a CHR program.

> Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime.

First of all, there are implicit facts in this proverb, namely, that there can be hungry men and that eating a fish saturates them. This is straightforwardly expressed in CHR as

$$r_1 \ @ \ \text{hungry}(M), \text{fish} \Leftrightarrow \text{saturated}(M).$$

Assuming days roughly correspond to rule applications, the above rule nicely captures the idea that you can give a fish to a man M in order to feed him. Of course, he eventually gets hungry again:

$$r_2 \ @ \ \text{saturated}(M) \Leftrightarrow \text{hungry}(M).$$

Now, we can consider the input $\langle \text{hungry}(M), \text{fish}; \emptyset; \top; \{M\}\rangle$, and unsurprisingly, this is a case of starvation – on the one hand, because we can only apply each of the two rules once, on the other hand, because the poor man has no more fish to eat.

Next, let us model what happens, if a man is taught to fish. We assume that the man is saturated, when we teach him to fish, and the result is clear: The man becomes a fisher.

$$r_3 \ @ \ \text{saturated}(M) \backslash \text{teach}(M) \Leftrightarrow \text{fisher}(M).$$

Of course, any fisher can follow his profession and go fishing. At this point, we make the forgivable assumption that our fisher is as extraordinary as the supply of fish, such that he can catch a fish whenever he wants to.

$$r_4 \ @ \ \text{fisher}(M) \Longrightarrow \text{fish}$$

We have now successfully modeled the second sentence of the above proverb: Given a man and someone willing to teach him to fish ($\langle \text{saturated}(M), \text{teach}(M); \emptyset; \top; \{M\}\rangle$), that man can feed himself for a lifetime (non-terminating computation).

Our model is quite realistic, as it also accounts for other phenomenons:

- What if the man refuses to be taught? By non-determinism and rule r_2, the man will become hungry, and at this point he has a serious starvation problem.
- What happens if one tries to teach a hungry man? As none of the rules is applicable in that case, he will simply starve. So we should at least bring one fish along to be safe.

Finally, when we consider several hungry men, we realize that our model solves the problem of nourishing the world's population: All we need is a single fisher – and if he is fair, he will even feed the immortals.

8.4 Merge Operator

In this section, we present the merge operator $\diamond$ that combines two CHR states. It is a beneficial tool in formal program analysis and different applications can be found in [Raiser, 2010]. Here, we analyze its properties in Section 8.4.1, before we discuss its implied partial order on states in Section 8.4.2. All results given in this section are formulated for $\omega_!$, but work equally for ω_e, when simply considering a projection of states that ignores the persistent store.

Definition 8.4.1 (Merge Operator $\diamond$)**.** *Let $\sigma_1 = \langle \mathbb{L}_1; \mathbb{P}_1; \mathbb{B}_1; \mathbb{V}_1 \rangle$ and $\sigma_2 = \langle \mathbb{L}_2; \mathbb{P}_2; \mathbb{B}_2; \mathbb{V}_2 \rangle$ such that local variables of one state are disjunct from all variables in the other state. Then for a set $\mathbb{V}$ of variables*

$$\sigma_1 \diamond_{\mathbb{V}} \sigma_2 ::= \langle \mathbb{L}_1 \uplus \mathbb{L}_2; \mathbb{P}_1 \uplus \mathbb{P}_2; \mathbb{B}_1 \wedge \mathbb{B}_2; (\mathbb{V}_1 \cup \mathbb{V}_2) \setminus \mathbb{V} \rangle.$$

We further lift this definition to equivalence classes. In that case, the merge operation assumes that two representants with accordingly disjunct variables are selected:

$$[\sigma_1] \diamond_{\mathbb{V}} [\sigma_2] ::= [\sigma_1 \diamond_{\mathbb{V}} \sigma_2].$$

For $\mathbb{V} = \emptyset$, we write $\sigma_1 \diamond \sigma_2$ and $[\sigma_1] \diamond [\sigma_2]$, respectively.

Applying the merge operator on equivalence classes assumes that two representants are selected that satisfy the above condition for their variables. Lemma 8.2.3 shows that renaming of local variables keeps equivalence of states, hence, such representants are guaranteed to exist.

Example 8.4.2. *Merging $\langle c(X); \emptyset; \top; \emptyset \rangle$ and $\langle \emptyset; \emptyset; X = 1; \emptyset \rangle$ should result in $\langle c(X); \emptyset; X = 1; \emptyset \rangle$. However, when considering equivalence classes we would instead get $[\langle c(X); \emptyset; \top; \emptyset \rangle]$, because $\langle \emptyset; \emptyset; X = 1; \emptyset \rangle \equiv_! \langle \emptyset; \emptyset; \top; \emptyset \rangle$.*

For that reason, the above definition restricts local variables to one state and allows turning global variables into local variables during the merge operation. Hence, we can perform the following merge operation:

$$[\langle c(X); \emptyset; \top; \{X\} \rangle] \diamond_{\{X\}} [\langle \emptyset; \emptyset; X = 1; \{X\} \rangle] = [\langle c(X); \emptyset; X = 1; \emptyset \rangle]$$

8.4.1 Properties of the Merge Operator

Lifting the merge operator $\diamond$ to equivalence classes is justified by the following lemma, which shows that it maintains state equivalence.

Lemma 8.4.3 ($\diamond_{\mathbb{V}}$ maintains Equivalence)**.** *Let $\sigma_1 \equiv_! \sigma_2$, then $(\sigma_1 \diamond_{\mathbb{V}} \tau) \equiv_! (\sigma_2 \diamond_{\mathbb{V}} \tau)$ for all $\mathbb{V}$.*

Proof. W.l.o.g. let $\sigma_i = \langle \mathbb{L}_i; \mathbb{P}_i; \mathbb{B}_i; \mathbb{V}' \rangle$ for $i = 1, 2$ and let $\tau = \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \hat{\mathbb{V}} \rangle$ such that the variables are disjunct according to Def. 8.4.1. Let $\bar{y}_1, \bar{y}_2$ be the local variables of σ_1 and σ_2 respectively. We know by Thm. 22 that:

$$\mathcal{CT} \models \begin{array}{l} \forall(\mathbb{B}_1 \rightarrow \exists \bar{y}_2.((\mathbb{L}_1 = \mathbb{L}_2) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_2)) \wedge \\ \forall(\mathbb{B}_2 \rightarrow \exists \bar{y}_1.((\mathbb{L}_1 = \mathbb{L}_2) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_1)) \end{array}$$

Let $\bar{x} = (\mathbb{V}' \cap \mathbb{V})$, then

$$\mathcal{CT} \models \begin{array}{l} \forall(\mathbb{B}_1 \rightarrow \exists \bar{y}_2 \exists \bar{x}.((\mathbb{L}_1 = \mathbb{L}_2) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_2)) \wedge \\ \forall(\mathbb{B}_2 \rightarrow \exists \bar{y}_1 \exists \bar{x}.((\mathbb{L}_1 = \mathbb{L}_2) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_1)) \end{array}$$

As $(\mathbb{L} = \mathbb{L})$ and $(\mathbb{P} \bowtie \mathbb{P})$ are tautologies, we can extend $(\mathbb{L}_1 = \mathbb{L}_2)$ to $((\mathbb{L}_1 \uplus \mathbb{L}) = (\mathbb{L}_2 \uplus \mathbb{L}))$ and analogously for $\mathbb{P}$. Similarly, $\mathbb{B} \to \mathbb{B}$ is a tautology, and therefore we have for $\bar{z}$ being the local variables of τ combined with $\hat{\mathbb{V}} \setminus \mathbb{V}$:

$$\mathcal{CT} \models \begin{array}{l} \forall(\mathbb{B}_1 \wedge \mathbb{B} \to \exists \bar{y}_2 \exists \bar{x} \exists \bar{z}.(((\mathbb{L}_1 \uplus \mathbb{L}) = (\mathbb{L}_2 \uplus \mathbb{L})) \wedge ((\mathbb{P}_1 \uplus \mathbb{P}) \bowtie (\mathbb{P}_2 \uplus \mathbb{P})) \wedge \mathbb{B}_2 \wedge \mathbb{B})) \wedge \\ \forall(\mathbb{B}_2 \to \exists \bar{y}_1 \exists \bar{x} \exists \bar{z}.(((\mathbb{L}_1 \uplus \mathbb{L}) = (\mathbb{L}_2 \uplus \mathbb{L})) \wedge ((\mathbb{P}_1 \uplus \mathbb{P}) \bowtie (\mathbb{P}_2 \uplus \mathbb{P})) \wedge \mathbb{B}_1 \wedge \mathbb{B})) \end{array}$$

As the local variables of $\sigma_1 \diamond_{\mathbb{V}} \tau$ are $\bar{x} \cup \bar{y}_1 \cup \bar{z}$, and analogously for $\sigma_2 \diamond_{\mathbb{V}} \tau$, we conclude by Thm. 22:

$$\sigma_1 \diamond_{\mathbb{V}} \tau = \langle \mathbb{L}_1 \uplus \mathbb{L}; \mathbb{P}_1 \uplus \mathbb{P}; \mathbb{B}_1 \wedge \mathbb{B}; (\mathbb{V}' \cup \hat{\mathbb{V}}) \setminus \mathbb{V} \rangle \equiv_! \langle \mathbb{L}_2 \uplus \mathbb{L}; \mathbb{P}_2 \uplus \mathbb{P}; \mathbb{B}_2 \wedge \mathbb{B}; (\mathbb{V}' \cup \hat{\mathbb{V}}) \setminus \mathbb{V} \rangle = \sigma_2 \diamond_{\mathbb{V}} \tau.$$

□

An important property for program analysis is monotonicity. The following lemma formulates monotonicity of CHR derivations using the merge operator.

Lemma 8.4.4 (Monotonicity)**.** *If* $[\sigma] \rightarrowtail_! [\tau]$ *then* $[\sigma] \diamond_{\mathbb{V}} [\sigma'] \rightarrowtail_! [\tau] \diamond_{\mathbb{V}} [\sigma']$ *for all* $\mathbb{V}$*.*

Proof. Let $[\sigma] \rightarrowtail_! [\tau]$ via rule r of the form $H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b$.

Case **ApplyLinear**: w.l.o.g. let $\sigma = \langle H_1^l \uplus H_2^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}_1 \rangle$. Then

$$[\sigma] \rightarrowtail_! [\langle H_1^l \uplus B_c \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}_1 \rangle] = [\tau]$$

and therefore for $\sigma' = \langle \mathbb{L}'; \mathbb{P}'; \mathbb{B}'; \mathbb{V}' \rangle$ with w.l.o.g. disjunct local variables:

$$\begin{array}{rl} & [\sigma] \diamond_{\mathbb{V}} [\sigma'] = [\sigma \diamond_{\mathbb{V}} \sigma'] \\ = & [\langle H_1^l \uplus H_2^l \uplus \mathbb{L} \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P} \uplus \mathbb{P}'; G \wedge \mathbb{B} \wedge \mathbb{B}'; (\mathbb{V}_1 \uplus \mathbb{V}') \setminus \mathbb{V} \rangle] \\ \rightarrowtail_! & [\langle H_1^l \uplus B_c \uplus \mathbb{L} \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P} \uplus \mathbb{P}'; G \wedge \mathbb{B} \wedge B_b \wedge \mathbb{B}'; (\mathbb{V}_1 \cup \mathbb{V}') \setminus \mathbb{V} \rangle] \\ = & [\tau \diamond_{\mathbb{V}} \sigma'] = [\tau] \diamond_{\mathbb{V}} [\sigma'] \end{array}$$

Case **ApplyPersistent**: w.l.o.g. let $\sigma = \langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}_1 \rangle$. Then

$$[\sigma] \rightarrowtail_! [\langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P} \uplus B_c; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}_1 \rangle] = [\tau]$$

and therefore for $\sigma' = \langle \mathbb{L}'; \mathbb{P}'; \mathbb{B}'; \mathbb{V}' \rangle$ with w.l.o.g. disjunct local variables:

$$\begin{array}{rl} & [\sigma] \diamond_{\mathbb{V}} [\sigma'] = [\sigma \diamond_{\mathbb{V}} \sigma'] \\ = & [\langle H_1^l \uplus \mathbb{L} \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P} \uplus \mathbb{P}'; G \wedge \mathbb{B} \wedge \mathbb{B}'; (\mathbb{V}_1 \uplus \mathbb{V}') \setminus \mathbb{V} \rangle] \\ \rightarrowtail_! & [\langle H_1^l \uplus \mathbb{L} \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P} \uplus B_c \uplus \mathbb{P}'; G \wedge \mathbb{B} \wedge B_b \wedge \mathbb{B}'; (\mathbb{V}_1 \cup \mathbb{V}') \setminus \mathbb{V} \rangle] \\ = & [\tau \diamond_{\mathbb{V}} \sigma'] = [\tau] \diamond_{\mathbb{V}} [\sigma'] \end{array}$$

□

By definition, the merge operator is commutative and closed on $\Sigma_!$. For $\mathbb{V} = \emptyset$, the merge operator $\diamond_{\mathbb{V}}$ is also associative on $\Sigma_!$, as the following lemma shows. Finally, the state $\sigma_\emptyset ::= \langle \emptyset; \emptyset; \top; \emptyset \rangle$ is the neutral element of $\diamond$ (for $\mathbb{V} = \emptyset$). In general however, $\sigma \diamond_{\mathbb{V}} \sigma_\emptyset \neq \sigma$, because σ may have global variables that also occur in $\mathbb{V}$.

Lemma 8.4.5. $(\Sigma_!/\equiv_!, \diamond)$ *is a commutative monoid.*

Proof. Totality of the merge operator and the neutral element $[\sigma_\emptyset]$ are clear. Commutativity is inherited from commutativity of the operators used in Def. 8.4.1, hence, it remains to show associativity of $\diamond$:

Let $\sigma_i = \langle \mathbb{L}_i; \mathbb{P}_i; \mathbb{B}_i; \mathbb{V}_i \rangle$ for $i = 1, 2, 3$, with w.l.o.g. suitably named variables, such that the below $\diamond$ operations are defined.

$\diamond$ on $\Sigma_!$:

$$\begin{aligned}
(\sigma_1 \diamond \sigma_2) \diamond \sigma_3 = \\
\langle \mathbb{L}_1 \uplus \mathbb{L}_2; \mathbb{P}_1 \uplus \mathbb{P}_2; \mathbb{B}_1 \wedge \mathbb{B}_2; \mathbb{V}_1 \cup \mathbb{V}_2 \rangle \diamond \sigma_3 = \\
\langle \mathbb{L}_1 \uplus \mathbb{L}_2 \uplus \mathbb{L}_3; \mathbb{P}_1 \uplus \mathbb{P}_2 \uplus \mathbb{P}_3; \mathbb{B}_1 \wedge \mathbb{B}_2 \wedge \mathbb{B}_3; \mathbb{V}_1 \cup \mathbb{V}_2 \cup \mathbb{V}_3 \rangle = \\
\sigma_1 \diamond \langle \mathbb{L}_2 \uplus \mathbb{L}_3; \mathbb{P}_2 \uplus \mathbb{P}_3; \mathbb{B}_2 \wedge \mathbb{B}_3; \mathbb{V}_2 \cup \mathbb{V}_3 \rangle = \\
\sigma_1 \diamond (\sigma_2 \diamond \sigma_3)
\end{aligned}$$

$\diamond$ on $\Sigma_!/\equiv_!$:

$$\begin{aligned}
([\sigma_1] \diamond [\sigma_2]) \diamond [\sigma_3] = \\
[\sigma_1 \diamond \sigma_2] \diamond [\sigma_3] = \\
[(\sigma_1 \diamond \sigma_2) \diamond \sigma_3] = \\
[\sigma_1 \diamond (\sigma_2 \diamond \sigma_3)] = \\
[\sigma_1] \diamond [\sigma_2 \diamond \sigma_3] = \\
[\sigma_1] \diamond ([\sigma_2] \diamond [\sigma_3])
\end{aligned}$$

□

Example 8.4.6. *For differing $\mathbb{V}$ and $\mathbb{V}'$ we have no associativity for the general case: $(\sigma_1 \diamond_\mathbb{V} \sigma_2) \diamond_{\mathbb{V}'} \sigma_3 \neq \sigma_1 \diamond_\mathbb{V} (\sigma_2 \diamond_{\mathbb{V}'} \sigma_3)$. This can be seen via the following example:*

$$\left.\begin{array}{ll} \sigma_1 = & \langle c(X); \emptyset; \top; \{X\} \rangle \\ \sigma_2 = & \langle \emptyset; \emptyset; \top; \emptyset \rangle \\ \sigma_3 = & \langle \emptyset; \emptyset; X = 1; \{X\} \rangle \\ \mathbb{V} = & \{X\} \\ \mathbb{V}' = & \emptyset \end{array}\right\} \Rightarrow \begin{array}{l} (\sigma_1 \diamond_\mathbb{V} \sigma_2) \diamond_{\mathbb{V}'} \sigma_3 = \ \textit{undefined} \\ \sigma_1 \diamond_\mathbb{V} (\sigma_2 \diamond_{\mathbb{V}'} \sigma_3) = \langle c(X); \emptyset; X = 1; \emptyset \rangle \\ ([\sigma_1] \diamond_\mathbb{V} [\sigma_2]) \diamond_{\mathbb{V}'} [\sigma_3] = [\langle c(Y); \emptyset; X = 1; \{X\} \rangle] \\ [\sigma_1] \diamond_\mathbb{V} ([\sigma_2] \diamond_{\mathbb{V}'} [\sigma_3]) = [\langle c(X); \emptyset; X = 1; \{X\} \rangle] \end{array}$$

Nevertheless, we have the following technical lemma:

Lemma 8.4.7. *Let $\sigma_1, \sigma_2, \sigma_3 \in \Sigma_!$ such that no local variable of a state occurs in another state. Then it holds for all $\mathbb{V}$ that*

$$\sigma_1 \diamond_\mathbb{V} (\sigma_2 \diamond \sigma_3) = (\sigma_1 \diamond \sigma_2) \diamond_\mathbb{V} \sigma_3$$

and

$$[\sigma_1] \diamond_\mathbb{V} ([\sigma_2] \diamond [\sigma_3]) = ([\sigma_1] \diamond [\sigma_2]) \diamond_\mathbb{V} [\sigma_3]$$

Proof. As all local variables of σ_1, σ_2, and σ_3 are disjunct, the second expression is well-defined and follows directly from the first. Therefore, only the first expression has to be shown. However, this follows directly from the associativity of the operators $\uplus, \cup$, and $\wedge$ used in the definition of operator $\diamond$ – observe that the global variables of the resulting representant state are $(\mathbb{V}_1 \cup \mathbb{V}_2 \cup \mathbb{V}_3) \setminus \mathbb{V}$ in both cases. □

8.4.2 Partial Order on States

As the merge operator (with $\mathbb{V} = \emptyset$) is a commutative monoid, there exists an implied preordering. Furthermore, it is also antisymmetric, hence, we arrive at the following partial order.

Lemma 8.4.8 (Partial Order $\lhd$)**.** *The commutative monoid* $(\Sigma_!/\equiv_!, \diamond)$ *implies a partial order* $\lhd$ *defined as follows, with* $\sigma, \sigma' \in \Sigma$*:*

$$[\sigma] \lhd [\sigma'] \text{ if and only if } \exists[\hat{\sigma}].[\sigma] \diamond [\hat{\sigma}] = [\sigma']$$

Proof. Every commutative monoid implies a preorder according to the above definition of $\lhd$. Therefore, it suffices to show antisymmetry:

Let $[\sigma_1] \lhd [\sigma_2]$ and $[\sigma_2] \lhd [\sigma_1]$, then there exist $[\hat{\sigma}_1]$ and $[\hat{\sigma}_2]$ as follows:

$$\left.\begin{array}{ll} [\sigma_2] = & [\sigma_1] \diamond [\hat{\sigma}_1] \\ [\sigma_1] = & [\sigma_2] \diamond [\hat{\sigma}_2] \end{array}\right\} \Rightarrow [\sigma_1] = ([\sigma_1] \diamond [\hat{\sigma}_1]) \diamond [\hat{\sigma}_2] = [\sigma_1] \diamond ([\hat{\sigma}_1] \diamond [\hat{\sigma}_2])$$

It follows that $[\hat{\sigma}_1] \diamond [\hat{\sigma}_2]$ equals $[\sigma_\emptyset]$, hence, $[\hat{\sigma}_1] = [\hat{\sigma}_2] = [\sigma_\emptyset]$ and therefore, $[\sigma_1] = [\sigma_2]$. □

However, special care has to be taken for states that contain local variables, because the resulting partial order may become counter-intuitive, as the following example shows.

Example 8.4.9. *Let* $\sigma_1 = \langle c(X); \emptyset; \top; \{X\}\rangle$*,* $\sigma_2 = \langle c(X); \emptyset; X = 1; \{X\}\rangle$*, then* $[\sigma_1] \lhd [\sigma_2]$ *(via* $\hat{\sigma} = \langle \emptyset; \emptyset; X = 1; \{X\}\rangle$*).*

However, for $\sigma_1' = \langle c(X); \emptyset; \top; \emptyset\rangle$ *and* $\sigma_2' = \langle c(X); \emptyset; X = 1; \emptyset\rangle$ *we do not have* $[\sigma_1'] \lhd [\sigma_2']$*. No* $\hat{\sigma}$ *can exist according to the definition of* $\lhd$*, because the variable restriction in Def. 8.4.1 for the merge operator* $\diamond$ *prohibits usage of the local variable* X *in* $\hat{\sigma}$*.*

This partial order combines well with monotonicity: Given that $[\sigma] \lhd [\sigma']$ and $[\sigma] \rightarrowtail^* [\tau]$, we have that $\exists[\hat{\sigma}].[\sigma] \diamond [\hat{\sigma}] = [\sigma']$ and $[\sigma'] \rightarrowtail^* [\tau] \diamond [\hat{\sigma}]$. Furthermore, we find that the neutral element $[\sigma_\emptyset]$ is also the least element of the partial order.

8.5 Discussion

8.5.1 Formulations of State Transition System

In this section, we provide an overview of different formulations available for the CHR state transition system. For both, ω_e and $\omega_!$, we identify three formulations that are so closely related, that we may freely switch between them. This gives us the freedom to select the most appropriate formulation at any time.

We begin with ω_e, for which Table 8.5 shows the different formulation possibilities for $(\Sigma_e, \rightarrowtail_e)$. For brevity, Table 8.5 and the following tables, omit additional textual requirements found in the corresponding definitions, for example, that we consider rule variants with a certain restriction on local variables, or that we assume local variables to be renamed such that the given merge operations are well-defined.

Variant I was our first formulation of the equivalence-based operational semantics, given in Definition 8.2.6. Its **Equivalence** rule resembles the compliance of state equivalence with rule application given by Theorem 21. Reducing the **Apply** rule, such that it is based on a rule state, yields **Variant** $I_\diamond$. While it makes **Apply** more compact, this comes at the cost of a third inference rule to handle merges. This **Merge** rule makes monotonicity explicitly part of the transition system (cf. Lemma 8.4.4). The implicitly required state splitting operation is discussed in more detail in [Raiser, 2010, Section 13.3].

Variant I**:** $(\Sigma_e, \rightarrowtail_e)$

Apply:
$$\frac{r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b}{\langle H_1 \uplus H_2 \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V} \rangle \rightarrowtail_e^r \langle H_1 \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V} \rangle}$$

Equivalence:
$$\frac{\sigma' \equiv \sigma \qquad \sigma \rightarrowtail^r \tau \qquad \tau \equiv \tau'}{\sigma' \rightarrowtail_e^r \tau'}$$

Variant $I_\diamond$**:** $(\Sigma_e, \rightarrowtail_e)$ with $\diamond$

Apply:
$$\frac{r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b}{\langle H_1 \uplus H_2; G; \mathbb{V} \rangle \rightarrowtail_e^r \langle H_1 \uplus B_c; G \wedge B_b; \mathbb{V} \rangle}$$

Equivalence:
$$\frac{\sigma' \equiv \sigma \qquad \sigma \rightarrowtail^r \tau \qquad \tau \equiv \tau'}{\sigma' \rightarrowtail_e^r \tau'}$$

Merge:
$$\frac{\sigma \rightarrowtail_e^r \tau}{\sigma \diamond_{\mathbb{V}} \delta \rightarrowtail_e^r \tau \diamond_{\mathbb{V}} \delta}$$

Table 8.5: Different Formulations of the Operational Semantics ω_e over Σ_e

Variant II**:** $(\Sigma_e/\equiv_e, \rightarrowtail_e)$

Apply:
$$\frac{r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b}{[\langle H_1 \uplus H_2 \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V} \rangle] \rightarrowtail_e^r [\langle H_1 \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V} \rangle]}$$

Variant $II_\diamond$**:** $(\Sigma_e/\equiv_e, \rightarrowtail_e)$ with $\diamond$

Apply:
$$\frac{r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b}{[\langle H_1 \uplus H_2; G; \mathbb{V} \rangle] \rightarrowtail_e^r [\langle H_1 \uplus B_c; G \wedge B_b; \mathbb{V} \rangle]}$$

Merge:
$$\frac{[\sigma] \rightarrowtail_e^r [\tau]}{[\sigma] \diamond_{\mathbb{V}} [\delta] \rightarrowtail_e^r [\tau] \diamond_{\mathbb{V}} [\delta]}$$

Table 8.6: Different Formulations of the Operational Semantics ω_e over $\Sigma_e/\equiv_e$

Abstracting over state equivalence yields **Variant** II in Table 8.6, which is based on equivalence classes. Due to directly rewriting the set of all equivalent states, it no longer requires the **Equivalence** rules from Table 8.5. Finally, **Variant** $II_\diamond$ analogously makes monotonicity explicit via the **Merge** rule.

Clearly, all formulations given in Table 8.5 and Table 8.6 are sound and complete with respect to each other and ω_{va}. For the remainder of this work, we will mostly apply **Variant** II, as it is the most succinct formulation. When discussing program analysis methods, however, we often refer to **Variant** $II_\diamond$ due to the importance of the merge operator in that context. We further found the direct rewriting of equivalence classes, instead of their single representant states, to be beneficial in the context of this work. Hence, the first two variants are rarely applied in the remaining chapters.

Analogously to Table 8.5 and Table 8.6, we list different formulations of the operational semantics $\omega_!$ in Table 8.7 and Table 8.8. We again find Theorem 21 in the form of the **Equivalence** rule and Lemma 8.4.4 corresponds to the **Merge** rule.

Variant I: $(\Sigma_!, \rightarrowtail_!)$

ApplyLinear:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p) \backslash (H_2^l \uplus H_2^p) \Leftrightarrow G \mid B_c, B_b \qquad H_2^l \neq \emptyset \qquad \sigma \not\equiv_! \tau}{\begin{array}{c}\sigma = \langle H_1^l \uplus H_2^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}\rangle \\ \rightarrowtail_!^r \langle H_1^l \uplus B_c \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}\rangle = \tau\end{array}}$$

ApplyPersistent:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p) \backslash H_2^p \Leftrightarrow G \mid B_c, B_b \qquad \sigma \not\equiv_! \tau}{\begin{array}{c}\sigma = \langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}\rangle \\ \rightarrowtail_!^r \langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus B_c \uplus \mathbb{P}; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}\rangle = \tau\end{array}}$$

Equivalence:
$$\frac{\sigma' \equiv \sigma \qquad \sigma \rightarrowtail_!^r \tau \qquad \tau \equiv \tau'}{\sigma' \rightarrowtail_e^r \tau'}$$

Variant $I_\diamond$: $(\Sigma_!, \rightarrowtail_!)$ with $\diamond$

ApplyLinear:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p) \backslash (H_2^l \uplus H_2^p) \Leftrightarrow G \mid B_c, B_b \qquad H_2^l \neq \emptyset \qquad \sigma \not\equiv_! \tau}{\begin{array}{c}\sigma = \langle H_1^l \uplus H_2^l; H_1^p \uplus H_2^p; G; \mathbb{V}\rangle \\ \rightarrowtail_!^r \langle H_1^l \uplus B_c; H_1^p \uplus H_2^p; G \wedge B_b; \mathbb{V}\rangle = \tau\end{array}}$$

ApplyPersistent:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p) \backslash H_2^p \Leftrightarrow G \mid B_c, B_b \qquad \sigma \not\equiv_! \tau}{\sigma = \langle H_1^l; H_1^p \uplus H_2^p; G; \mathbb{V}\rangle \rightarrowtail_!^r \langle H_1^l; H_1^p \uplus H_2^p \uplus B_c; G \wedge B_b; \mathbb{V}\rangle = \tau}$$

Equivalence:
$$\frac{\sigma' \equiv \sigma \qquad \sigma \rightarrowtail_!^r \tau \qquad \tau \equiv \tau'}{\sigma' \rightarrowtail_e^r \tau'}$$

Merge:
$$\frac{\sigma \rightarrowtail_!^r \tau}{\sigma \diamond \delta \rightarrowtail_!^r \tau \diamond \delta}$$

Table 8.7: Different Formulations of the Operational Semantics $\omega_!$ over $\Sigma_!$

Variant II: $(\Sigma_!/\equiv_!, \rightarrowtail_!)$

ApplyLinear:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p)\backslash(H_2^l \uplus H_2^p) \Leftrightarrow G \mid B_c, B_b \qquad H_2^l \neq \emptyset \qquad [\sigma] \neq [\tau]}{\begin{array}{c}[\sigma] = [\langle H_1^l \uplus H_2^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}\rangle] \\ \rightarrowtail_!^r [\langle H_1^l \uplus B_c \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}\rangle] = [\tau]\end{array}}$$

ApplyPersistent:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p)\backslash H_2^p \Leftrightarrow G \mid B_c, B_b \qquad [\sigma] \neq [\tau]}{\begin{array}{c}[\sigma] = [\langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus \mathbb{P}; G \wedge \mathbb{B}; \mathbb{V}\rangle] \\ \rightarrowtail_!^r [\langle H_1^l \uplus \mathbb{L}; H_1^p \uplus H_2^p \uplus B_c \uplus \mathbb{P}; G \wedge \mathbb{B} \wedge B_b; \mathbb{V}\rangle] = [\tau]\end{array}}$$

Variant $II_\diamond$: $(\Sigma_!/\equiv_!, \rightarrowtail_!)$ with $\diamond$

ApplyLinear:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p)\backslash(H_2^l \uplus H_2^p) \Leftrightarrow G \mid B_c, B_b \qquad H_2^l \neq \emptyset \qquad [\sigma] \neq [\tau]}{\begin{array}{c}[\sigma] = [\langle H_1^l \uplus H_2^l; H_1^p \uplus H_2^p; G; \mathbb{V}\rangle] \\ \rightarrowtail_!^r [\langle H_1^l \uplus B_c; H_1^p \uplus H_2^p; G \wedge B_b; \mathbb{V}\rangle] = [\tau]\end{array}}$$

ApplyPersistent:
$$\frac{r \mathbin{@} (H_1^l \uplus H_1^p)\backslash H_2^p \Leftrightarrow G \mid B_c, B_b \qquad [\sigma] \neq [\tau]}{\begin{array}{c}[\sigma] = [\langle H_1^l; H_1^p \uplus H_2^p; G; \mathbb{V}\rangle] \\ \rightarrowtail_!^r [\langle H_1^l; H_1^p \uplus H_2^p \uplus B_c; G \wedge B_b; \mathbb{V}\rangle] = [\tau]\end{array}}$$

Merge:
$$\frac{[\sigma] \rightarrowtail_!^r [\tau]}{[\sigma] \diamond_{\mathbb{V}} [\delta] \rightarrowtail_!^r [\tau] \diamond_{\mathbb{V}} [\delta]}$$

Table 8.8: Different Formulations of the Operational Semantics $\omega_!$ over $\Sigma_!/\equiv_!$

8.5.2 Range-Restrictedness

As specified in Definition 8.3.5, $\omega_!$ requires range-restricted programs. In the following, we explain why a naive extension to the full segment of CHR by dropping this restriction would violate soundness.

We recall that a persistent constraint is a finite representation of an arbitrary number of identical constraints, as generated under ω_e by a propagation rule from the range-restricted segment. Under the same conditions, however, a propagation rule with local variables would generate an arbitrary number of nearly but *not quite* identical constraints, as the local variables would be renamed apart between any two of those nearly identical constraints, which the following example demonstrates.

Example 8.5.1. *Consider the following program:*

$$\begin{array}{llll} r_1 & @ & a & \Longrightarrow \quad b(X) \\ r_2 & @ & b(X), b(X) & \Leftrightarrow \quad c \end{array}$$

When executed with an initial a-constraint, this program causes the following infinite derivation under ω_e.

$$\begin{array}{ll} & [\langle a; \top; \emptyset\rangle] \\ \rightarrowtail_e^{r_1} & [\langle a, b(X'); \top; \emptyset\rangle] \\ \rightarrowtail_e^{r_1} & [\langle a, b(X'), b(X''); \top; \emptyset\rangle] \rightarrowtail_e^{r_1} \ldots \end{array}$$

The variables $X', X'', \ldots$ are distinct from each other and from the variable X which occurs in the rule body. Thus, it is impossible to derive the c-constraint from the a-constraint under ω_e.

Under the current approach, we cannot finitely represent an arbitrary number of such nearly identical constraints. A naive extension of $\omega_!$ to the full segment of CHR as specified above would discard the distinction between the two types of generated constraints altogether.

Example 8.5.2. *With respect to the previous example, a naive extension of $\omega_!$ would make the following derivation possible:*

$$\begin{array}{ll} & [\langle a; \emptyset; \top; \emptyset\rangle] \\ \rightarrowtail_!^{r_1} & [\langle a; b(X'); \top; \emptyset\rangle] = [\langle a; b(X'), b(X'); \top; \emptyset\rangle] \\ \rightarrowtail_!^{r_2} & [\langle a; b(X'), b(X'), c; \top; \emptyset\rangle] \end{array}$$

As the above example showed, simply applying Definition 8.3.5 to non-range-restricted programs results in a loss of soundness.

8.5.3 Termination Behavior

The operational semantics $\omega_!$ is an approach to treating propagation rules in CHR that is completely different from the previous token-based approaches. This especially results in a differing termination behavior. There exist programs that terminate in $\omega_!$ but not in ω_t and ω_p, and vice versa.

Example 8.5.3. *Consider again the example program for computing the transitive hull, given in Example 8.3.1. Due to the presence of a propagation rule, it is non-terminating under ω_e. Under ω_t and ω_p, termination depends on the initial goal: It is shown in [Pilozzi and De Schreye, 2009] that it terminates for acyclic graphs. However, goals containing cyclic graphs, such as $\langle e(1,2), e(2,1); \emptyset; \top; \emptyset\rangle_0^\emptyset$, entail non-terminating behavior. The following derivation is not exact according to ω_t, but uses simplified states for better readability.*

$$\begin{array}{ll} & \langle e(1,2), e(2,1); \emptyset; \top; \emptyset\rangle_0^\emptyset \\ \rightarrowtail_t^* & \langle \emptyset; e(1,2)\#0, e(2,1)\#1; \top; \emptyset\rangle_2^\emptyset \\ \rightarrowtail_t & \langle e(1,1); e(1,2)\#0, e(2,1)\#1; \top; \{(t,0,1)\}\rangle_2^\emptyset \\ \rightarrowtail_t & \langle \emptyset; e(1,2)\#0, e(2,1)\#1, e(1,1)\#2; \top; \{(t,0,1)\}\rangle_3^\emptyset \\ \rightarrowtail_t & \langle e(1,2); e(1,2)\#0, e(2,1)\#1, e(1,1)\#2; \top; \{(t,0,1),(t,2,0)\}\rangle_3^\emptyset \\ \rightarrowtail_t & \ldots \end{array}$$

Under $\omega_!$, the previous goal terminates after computing the transitive hull.

$$\begin{array}{ll} & [\langle \{e(1,2), e(2,1)\}; \emptyset; \top; \emptyset\rangle] \\ \rightarrowtail_! & [\langle \{e(1,2), e(2,1)\}; \{e(1,1)\}; \top; \emptyset\rangle] \\ \rightarrowtail_!^* & [\langle \{e(1,2), e(2,1)\}; \{e(1,1), e(1,2), e(2,1), e(2,2)\}; \top; \emptyset\rangle] \not\rightarrowtail_! \end{array}$$

The transitive hull program benefits from execution under $\omega_!$. While it is the most natural formulation of a transitivity property in terms of a CHR rule, current implementations cannot use it in that form due to its non-terminating behavior on circular input graphs. For $\omega_!$, we can instead prove termination for arbitrary inputs.

Lemma 8.5.4. *Under $\omega_!$, the transitive hull program terminates for every possible input.*

Proof. The only rule propagates constraints of type $e/2$, which necessarily become persistent. The propagated constraints contain only the arguments X, Z, received as arguments

in the rule head. Hence, no new arguments are introduced. Any given initial state contains a finite number of arguments used in $e/2$ constraints. From these, only finitely many different $e/2$ constraints can be built. As rule application is irreflexive, the computation therefore has to stop after a finite number of transition steps. □

Nevertheless, program termination in $\omega_!$ is not strictly stronger than that in ω_t or ω_p, as the following counterexample shows.

Example 8.5.5. *Consider the following exemplary CHR program.*

$$\begin{array}{llll} r_1 & @ & a & \Longrightarrow \quad b \\ r_2 & @ & c(X), b & \Leftrightarrow \quad c(X{+}1) \end{array}$$

The program terminates in ω_t (and ω_p): As there can only be a finite number of a-constraints in the initial goal, rule r_1 will create a finite number of b-constraints as well. These will be consumed by rule r_2 in finite time, followed by quiescence. We again simplified the following states for better readability.

$$\begin{array}{ll} & \langle a, c(X); \emptyset; \top; \emptyset \rangle_0^{\{X\}} \\ \rightarrowtail_t^* & \langle \emptyset; a\#0, b\#1, c(X)\#2; \top; (r_1, 0) \rangle_3^{\{X\}} \\ \rightarrowtail_t^{r_2} & \langle c(X{+}1); a\#0; \top; (r_1, 0) \rangle_3^{\{X\}} \\ \rightarrowtail_t & \langle \emptyset; a\#0, c(X{+}1)\#3; \top; (r_1, 0) \rangle_4^{\{X\}} \not\rightarrowtail_t \end{array}$$

In contrast, the same program exhibits non-terminating behavior in $\omega_!$, as the following infinite derivation shows:

$$\begin{array}{ll} & [\langle a, c(X); \emptyset; \top; \{X\} \rangle] \\ \rightarrowtail_!^{r_1} & [\langle a, c(X); b; \top; \{X\} \rangle] \\ \rightarrowtail_!^{r_2} & [\langle a, c(X{+}1); b; \top; \{X\} \rangle] \\ \rightarrowtail_!^{r_2} & [\langle a, c(X{+}2); b; \top; \{X\} \rangle] \\ \rightarrowtail_!^{r_2} & \ldots \end{array}$$

This difference in termination behavior is neither good nor bad. It only shows that our approach of persistent constraints is an entirely different one from the token-based approaches. Therefore, programs developed for either operational semantics should not naively be executed in the other, lest results may change unexpectedly.

8.5.4 Expressivity

In this section we compare expressivity of the operational semantics $\omega_e, \omega_t, \omega_p$, and $\omega_!$. Section 8.5.4 first introduces how we formally compare expressivity of different operational semantics, before Section 8.5.4 presents the results of our comparison.

Expressivity of Operational Semantics

As all of the compared operational semantics are Turing-complete [Sneyers et al., 2009], expressivity is compared in the literature via the concept of acceptable encoding. This concept originates from Shapiro [Shapiro, 1989] and was first applied to CHR by Gabbrielli et al. [2009]. It relies on the notion of *answer* defined below.

In order to distinguish linear and persistent constraints when considering goals, we introduce for each CHR constraint symbol c/n, denoting a linear constraint, a corresponding

fresh symbol $!c/n$, denoting a persistent constraint. For a multiset $M = \{c_1(\bar{t}_1), \ldots, c_n(\bar{t}_n)\}$ let $!M = \{!c_1(\bar{t}_1), \ldots, !c_n(\bar{t}_n)\}$.

In the literature answers are usually defined as logical formulas, expressing the declarative reading of a final state. We found it more suitable to define them as ω_e states for two reasons: Firstly, unlike logical formulas, ω_e states are aware of multiplicities of constraints. Secondly, ω_e states enable us to exploit $\equiv_e$ when comparing answers.

Definition 8.5.6 (Answers)**.** *Let $G \wedge B$ be a goal with CHR constraints G and built-in constraints B. Then the set of equivalence classes of ω_e states $\mathcal{A}_\mathcal{P}(G \wedge B)$ for a program $\mathcal{P}$ is called the (set of)* answers *and is defined as follows:*

- *for ω_e:* $\mathcal{A}^e_\mathcal{P}(G \wedge B) = \{[\tau] \mid [\langle G; B; \text{vars}(G \wedge B)\rangle] \rightarrowtail_e^* [\tau] \not\rightarrowtail_e\}$
- *for ω_t:* $\mathcal{A}^t_\mathcal{P}(G \wedge B) = \{[\langle \text{chr}(\mathbb{G}); \mathbb{B}; \text{vars}(G \wedge B)\rangle] \mid \langle G, B; \emptyset; \top; \emptyset\rangle_0^{\text{vars}(G \wedge B)} \rightarrowtail_t^* \langle \emptyset; \mathbb{G}; \mathbb{B}; T\rangle_n^{\text{vars}(G \wedge B)} \not\rightarrowtail_t\}$
- *for ω_p: $\mathcal{A}^p$ is defined analogously to $\mathcal{A}^t$.*
- *for $\omega_!$:* $\mathcal{A}^!_\mathcal{P}(G \wedge B) = \{[\langle \mathbb{L} \uplus !\mathbb{P}; \mathbb{B}; \text{vars}(G \wedge B)\rangle] \mid G = L \uplus !P \wedge \langle L; P; B; \text{vars}(G \wedge B)\rangle \rightarrowtail_!^* \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \text{vars}(G \wedge B)\rangle \not\rightarrowtail_!\}$

The following definition is based on the definition of Gabbrielli et al. [2009] for an acceptable encoding for CHR operational semantics.

Definition 8.5.7 (Acceptable Encoding)**.** *Let ω_1, ω_2 be two operational semantics, $\mathcal{P}_i$ the set of all ω_i programs, and $\mathcal{G}_i$ the set of all ω_i goals for $i = 1, 2$. An* acceptable encoding *of ω_1 into ω_2 is a pair of mappings $[\![\ ]\!] : \mathcal{P}_1 \to \mathcal{P}_2$ and $[\![\ ]\!]_g : \mathcal{G}_1 \to \mathcal{G}_2$ which satisfy the following conditions:*

- *$\mathcal{P}_1$ and $\mathcal{P}_2$ share the same constraint theory $\mathcal{CT}$;*
- *for any goal $G \in \mathcal{G}_1$ let $c, d \in G$ be CHR constraints, then $[\![c, d]\!]_g = [\![c]\!]_g \uplus [\![d]\!]_g$. For any built-in constraint $b \in G$ we have $[\![b]\!]_g = b$.*
- *Answers are preserved, i.e., $\forall G \in \mathcal{G}_1 . \forall \mathcal{P} \in \mathcal{P}_1 . \mathcal{A}^2_{[\![\mathcal{P}]\!]}([\![G]\!]_g) = [\![\mathcal{A}^1_\mathcal{P}(G)]\!]_g$ holds.*

Comparison Results

Figure 8.1 orders the different operational semantics by expressivity. As shown by Gabbrielli et al. [2009], there exists an acceptable encoding to embed ω_t into ω_p, but not vice versa. Thus, ω_p is strictly more expressive than ω_t, denoted by the corresponding arrow in Figure 8.1. In this work, we furthermore show that ω_p is strictly more expressive than $\omega_!$ and that ω_e is strictly less expressive than both ω_t and $\omega_!$.

Concerning the embedding of ω_e into $\omega_!$, we assume range-restricted programs only. Concerning the acceptable encodings of $\omega_!$ into ω_t and ω_p, we require that the respective programs do not contain pathological rules, according to the following definition.

Definition 8.5.8 (Pathological Rules)**.** *A CHR rule*

$$r \ @ \ H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b$$

is called pathological *if and only if*

$$\exists \mathbb{B} . \langle H_2; \mathbb{B} \wedge G; \emptyset\rangle \equiv_e \langle B_c; B_b; \emptyset\rangle$$

It is called trivially pathological *iff $\mathbb{B} = \top$. A CHR program $\mathcal{P}$ is called pathological if it contains at least one pathological rule.*

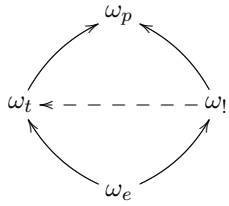


Figure 8.1: Acceptable encodings between different operational semantics

The range-restriction requirement on ω_e programs is due to the fact that Definition 8.3.5 for $\omega_!$ is only defined on range-restricted programs. The restriction to non-pathological programs for embeddings of $\omega_!$ into ω_t and ω_p ensures **ApplyLinear** transitions never fail due to irreflexivity, according to the following Lemma.

Concerning the relationship of ω_t and $\omega_!$, we found that no acceptable encoding of ω_t into $\omega_!$ exists. We did find an acceptable encoding of $\omega_!$ into ω_t. However, a thusly encoded program might exhibit a different termination behavior from the original $\omega_!$ program (cf. Example 8.5.16), as visualized by the dashed arrow in Figure 8.1. We currently do not know whether an acceptable encoding without that limitation exists.

The definition of pathological rules is chosen such as to coincide with those rules that cause redundant rule applications – modulo state equivalence – in ω_e.

Lemma 8.5.9. *Let $\mathcal{P}$ be a non-pathological CHR program. Then for all ω_e states $\sigma, \tau \in \Sigma_e$ where $[\sigma] \rightarrowtail_e [\tau]$, we have $\sigma \not\equiv_e \tau$.*

Proof. We first show a property of Def. 8.5.8: Let $\langle H_2; \mathbb{B} \wedge G; \emptyset\rangle \equiv_e \langle B_c; B_b; \emptyset\rangle$, w.l.o.g. let the respective local variables $\bar{y}, \bar{y}'$ be renamed apart. Then by Thm. 20:

$$\mathcal{CT} \models \forall(\mathbb{B} \wedge G \rightarrow \exists\bar{y}'.((H_2 = B_c) \wedge B_b)) \text{ and}$$
$$\mathcal{CT} \models \forall(B_b \rightarrow \exists\bar{y}.((H_2 = B_c) \wedge \mathbb{B} \wedge G)$$

This is logically equivalent to

$$\mathcal{CT} \models \forall(\mathbb{B} \wedge G \rightarrow \exists\bar{y}'.((H_2 = B_c) \wedge B_b \wedge \mathbb{B} \wedge G)) \text{ and}$$
$$\mathcal{CT} \models \forall(B_b \wedge \mathbb{B} \wedge G \rightarrow \exists\bar{y}.((H_2 = B_c) \wedge \mathbb{B} \wedge G))$$

Therefore, again by Thm. 20, we have that

$$\langle H_2; G \wedge \mathbb{B}; \emptyset\rangle \equiv_e \langle B_c; B_b; \emptyset\rangle \equiv_e \langle B_c; B_b \wedge G \wedge \mathbb{B}; \emptyset\rangle$$

Now let r be a rule $r \ @\ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b$ such that $[\sigma] \rightarrowtail_e^r [\tau]$. It follows that $\sigma \equiv_e \langle H_1 \uplus H_2 \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V}\rangle$ and $\tau \equiv_e \langle B_c \uplus H_1 \uplus \mathbb{G}; B_b \wedge G \wedge \mathbb{B}; \mathbb{V}\rangle$.

Assume that $\sigma \equiv_e \tau$. As H_1 and $\mathbb{G}$ occur in both states, the corresponding states with those multisets removed are also equivalent. Similarly, the same states with $\emptyset$ instead of $\mathbb{V}$ for global variables are equivalent. Therefore,

$$\langle H_2; G \wedge \mathbb{B}; \emptyset\rangle \equiv_e \langle B_c; B_b \wedge G \wedge \mathbb{B}; \emptyset\rangle$$

This implies that there exists a $\mathbb{B}$ according to Def. 8.5.8, which is a contradiction to the program being non-pathological. Hence, $\sigma \not\equiv_e \tau$. □

The following lemmata are proofs for the arrows present in Figure 8.1.

Lemma 8.5.10 ($\omega_t \to \omega_p$)**.** *There exists an acceptable encoding of ω_t into ω_p.*

Proof (sketch). Set all rules to priority 1. □

Lemma 8.5.11 ($\omega_p \not\to \omega_t$)**.** *There exists no acceptable encoding of ω_p into ω_t.*

Proof. Follows directly from Gabbrielli et al. [2009]. □

Lemma 8.5.12 ($\omega_e \to \omega_t$)**.** *There exists an acceptable encoding of ω_e into ω_t.*

Proof (sketch). Replace propagation rules with simplification rules that contain a copy of the head in their bodies. □

Lemma 8.5.13 ($\omega_t \not\to \omega_e$)**.** *There exists no acceptable encoding of ω_t into ω_e.*

Proof. For any program $\mathcal{P}'$ if $\sigma = \langle G'; B'; \emptyset\rangle$ with $[\sigma] \in \mathcal{A}^e_{\mathcal{P}'}(G)$, no rule in $\mathcal{P}'$ is applicable to $[\sigma]$. Consider the ω_t program $\mathcal{P} = (a \Rightarrow b)$. Since $\mathcal{A}^t_{\mathcal{P}}(a) = \{[\langle a, b; \top; \emptyset\rangle]\}$ and $\mathcal{A}^t_{\mathcal{P}}(a, b) = \{[\langle a, b, b; \top; \emptyset\rangle]\}$, an acceptable encoding has to satisfy $\mathcal{A}^e_{[\![\mathcal{P}]\!]}([\![a]\!]_g) = \{[\langle [\![a, b]\!]_g; \top; \emptyset\rangle]\}$ and $\mathcal{A}^e_{[\![\mathcal{P}]\!]}([\![a, b]\!]_g) = \{[\langle [\![a, b, b]\!]_g; \top; \emptyset\rangle]\} = \{[\langle [\![a]\!]_g \uplus [\![b]\!]_g \uplus [\![b]\!]_g; \top; \emptyset\rangle]\} \neq \{[\langle [\![a]\!]_g \uplus [\![b]\!]_g; \top; \emptyset\rangle]\} = \{[\langle [\![a, b]\!]_g; \top; \emptyset\rangle]\}$ which contradicts our earlier observation. □

Lemma 8.5.14 ($\omega_t \not\to \omega_!$)**.** *There exists no acceptable encoding of ω_t into $\omega_!$.*

Proof. Analogously to Lemma 8.5.13.

□

Lemma 8.5.15 ($\omega_! \to \omega_t$)**.** *There exists an acceptable encoding of $\omega_!$ into ω_t.*

Proof. We show how to encode any $\omega_!$ program $\mathcal{P}$ in ω_t. For every n-ary constraint c/n in $\mathcal{P}$, there exists an $(n+1)$-ary constraint $c/n+1$ in the encoding. In the following, for a multiset of user-defined $\omega_!$-constraints $M = \{c_1(\bar{t}_1), \ldots, c_n(\bar{t}_n)\}$ let $l(M) ::= \{c_1(l, \bar{t}_1), \ldots, c_n(l, \bar{t}_n)\}$ and $p(M) ::= \{c_1(p, \bar{t}_1), \ldots, c_n(p, \bar{t}_n)\}$.

The encoded program $[\![\mathcal{P}]\!]$ is constructed as follows:

1. For every rule $r \ @\ H_1 \setminus H_2 \Leftrightarrow G \mid B$ in $\mathcal{P}$, and all multisets $H^l_1, H^p_1, H^l_2, H^p_2$ s.t. $H^l_1 \uplus H^p_1 = H_1$ and $H^l_2 \uplus H^p_2 = H_2$ and $H^l_2 \neq \emptyset$, the following rule is in $[\![\mathcal{P}]\!]$:

$$l(H^l_1) \uplus p(H^p_1) \uplus p(H^p_2) \setminus l(H^l_2) \Leftrightarrow G \mid l(B_c), B_b$$

2. For every rule $r \ @\ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b$ in $\mathcal{P}$, and all multisets H^l_1, H^p_1 s.t. $H^l_1 \uplus H^p_1 = H_1$, the following rule is in $[\![\mathcal{P}]\!]$:

$$l(H^l_1) \uplus p(H^p_1) \uplus p(H_2) \Rightarrow G \mid p(B_c), B_b$$

3. For every rule $\{c(p, \bar{t}), c(p, \bar{t}')\} \uplus H_1 \setminus H_2 \Leftrightarrow G \mid B$ in $[\![\mathcal{P}]\!]$, add also the following rule:

$$\{c(p, \bar{t})\} \uplus H_1 \setminus H_2 \Leftrightarrow \bar{t} = \bar{t}' \wedge G \mid B$$

4. For every user-defined constraint declaration c_n in $\mathcal{P}$, there is a rule

$$c(p, \bar{t}) \setminus c(p, \bar{t}) \Leftrightarrow \top$$

The translation of goals is defined as:

$$[\![\mathbb{L} \uplus !\mathbb{P}]\!]_g ::= l(\mathbb{L}) \uplus p(\mathbb{P})$$

Soundness: Let $\mathcal{S}_!$ be a function mapping from ω_t states to $\Sigma_!$ such that for $\sigma_t = \langle l(\mathbb{L}) \uplus p(\mathbb{P}) \uplus \mathbb{B}'; \mathbb{S}; \mathbb{B}; \mathbb{T} \rangle_k^{\mathbb{V}}$ where $\mathrm{chr}(\mathbb{S}) = l(\mathbb{L}') \uplus p(\mathbb{P}')$ for some $\mathbb{L}', \mathbb{P}'$,

$$\mathcal{S}_!(\sigma_t) ::= [\langle \mathbb{L} \uplus \mathbb{L}'; \mathbb{P} \uplus \mathbb{P}'; \mathbb{B} \wedge \mathbb{B}'; \mathbb{V} \rangle]$$

In the following, we will show that for all $\sigma_t, \tau_t \in \Sigma_t$, $\sigma_t \rightarrowtail_t^* \tau_t$ implies $\mathcal{S}_!(\sigma_t) \rightarrowtail_!^* \mathcal{S}_!(\tau_t)$.

It is clear from the definition that both the **Introduce** and **Solve** transitions of ω_t are invariant to the $\mathcal{S}_!$ function. Concerning **Apply**, we proceed stepwise w.r.t. the application of the four types of rules present in the encoding $[\![\mathcal{P}]\!]$.

1. The rules introduced in construction step 1 represent **ApplyLinear** transitions in $\mathcal{P}$.

Let r be a variant of a rule $l(H_1^l) \uplus p(H_1^p) \uplus p(H_2^p) \setminus l(H_2^l) \Leftrightarrow G \mid l(B_c) \uplus B_b$ in $[\![\mathcal{P}]\!]$ with fresh variables $\bar{y}$. By definition of the encoding, r has a corresponding rule $r' \mathbin{@} H_1^l \uplus H_1^p \setminus H_2^l \uplus H_2^p \Leftrightarrow G \mid B_c \uplus B_b$ in $\mathcal{P}$. We assume w.l.o.g. that the goal store of σ_t is empty. Hence let $\sigma_t = \langle \emptyset; l(\mathbb{L}) \uplus p(\mathbb{P}); \mathbb{B}; \mathbb{T} \rangle_k^{\mathbb{V}}$ and assume that $\sigma_t \rightarrowtail_t^r \tau_t$. From Def. 1.1.5 follows that $\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}.(l(H_1^l) \uplus l(H_2^l) \uplus l(\mathbb{L}') = l(\mathbb{L}) \wedge p(H_1^p) \uplus p(H_2^p) \uplus p(\mathbb{P}') = p(\mathbb{P}) \wedge G))$ for some $\mathbb{L}', \mathbb{P}'$. Hence, $\mathcal{S}_!(\sigma_t) = [\langle H_1^l \uplus H_2^l \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P}'; (H_1^l \uplus H_2^l \uplus \mathbb{L}' = \mathbb{L}) \wedge (H_1^p \uplus H_2^p \uplus \mathbb{P}' = \mathbb{P}) \wedge G \wedge \mathbb{B}; \mathbb{V} \rangle]$. Using Def. 1.1.5 and Def. 8.3.5, we now have that $\mathcal{S}_!(\sigma_t) \rightarrowtail_!^{r'} \mathcal{S}_!(\tau_t)$ or $\mathcal{S}_!(\sigma_t) = \mathcal{S}_!(\tau_t)$.

2. The rules introduced in step 2 represent **ApplyPersistent** transitions. Analogously to step 1, we have that $\sigma_t \rightarrowtail_t^r \tau_t$ implies $\mathcal{S}_!(\sigma_t) \rightarrowtail_!^{r'} \mathcal{S}_!(\tau_t)$ for some rule $r' \in \mathcal{P}$ or $\mathcal{S}_!(\sigma_t) = \mathcal{S}_!(\tau_t)$.

3. Step 3 introduces further rules for both **ApplyLinear** and **ApplyPersistent** transitions where a single persistent constraint in the store matches with several head constraints.

For example, the rule $c(X), c(Y) \Leftrightarrow d(X, Y)$ is applicable to the state $[\langle \emptyset; c(0); \top; \emptyset \rangle]$ in $\omega_!$, since $\langle \emptyset; c(0); \top; \emptyset \rangle \equiv_! \langle \emptyset; c(0), c(0); \top; \emptyset \rangle$. Step 2 of the embedding introduces the rule $c(p, X), c(p, Y) \Leftrightarrow d(p, X, Y)$ and step 3 furthermore introduces $c(p, X) \Leftrightarrow X = Y | d(p, X, Y)$, which matches with the ω_t state $\langle \emptyset; c(p, 0); \top; \emptyset \rangle_k^{\mathbb{V}}$. Strengthening of the guard might result in a redundant rule: For the rule $c(X), c(Y) \Leftrightarrow X > Y \mid d(X, Y)$, the rule $c(p, X) \Leftrightarrow \bot | d(p, X, Y)$ is introduced which cannot be fired by definition.

To proof soundness, let $\sigma = \langle \mathbb{L}; \{c(p, \bar{t}), c(p, \bar{t}')\} \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle$ and $\sigma' = \langle \mathbb{L}; \{c(p, \bar{t})\} \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle$ such that $[\sigma] \rightarrowtail_!^r [\tau]$ for some τ. If $\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \bar{t} = \bar{t}')$, we have $\sigma \equiv_! \sigma'$, and hence $[\sigma'] \rightarrowtail_!^r [\tau]$. The soundness of the rules introduced in step 3 is thus reduced to the soundness of those from step 1 and step 2.

4. The rules introduced in step 4 enforce a minimal representation of the persistent store. As $\mathcal{S}_!(\langle \mathbb{G}; \{c(\bar{t}), c(\bar{t})\} \uplus \mathbb{P}; \mathbb{B}; \mathbb{T} \rangle_k^{\mathbb{V}}) = \mathcal{S}_!(\langle \mathbb{G}; \{c(\bar{t})\} \uplus \mathbb{P}; \mathbb{B}; \mathbb{T} \rangle_k^{\mathbb{V}})$, they are invariant to soundness.

From 1-4 follows that $\sigma_t \rightarrowtail_t^* \tau_t$ implies $\mathcal{S}_!(\sigma_t) \rightarrowtail_!^* \mathcal{S}_!(\tau_t)$.

Now assume that τ_t is a *fixed point* w.r.t. $[\![\mathcal{P}]\!]$. This implies that for every possible match (if any) between sequences of constraints H_1, H_2 in τ_t and a rule r in $[\![\mathcal{P}]\!]$, there is a token $(r, id(H_1) + id(H_2))$ in the propagation history $\mathbb{T}_\tau$ inhibiting the firing of r. It follows that r is of the form $r \mathbin{@} l(H_1^l) \uplus p(H^p) \Rightarrow G \mid p(B_c) \uplus B_b$ and that $p(B_c)$ and B_b are already contained in τ_t from an earlier firing of r. Hence, for every possible match (if any) between constraints in $\mathcal{S}_!(\tau_t)$ and a rule r' in $\mathcal{P}$, the firing of r' is inhibited by the irreflexivity condition. Thus, $\mathcal{S}_!(\tau_t)$ also is a fixed point w.r.t. $\mathcal{P}$.

So finally, if from $\sigma_t = \langle l(\mathbb{L}) \uplus p(\mathbb{P}) \uplus \mathbb{B}; \emptyset; \top; \emptyset\rangle_0^{\mathbb{V}}$ we can derive a fixed point $\tau_t = \langle \emptyset; \mathbb{S}; \mathbb{B}'; \mathbb{T}\rangle_n^{\mathbb{V}}$ where $\mathrm{chr}(\mathbb{S}) = l(\mathbb{L}') \uplus p(\mathbb{P}')$, then from the $\omega_!$ state $[\langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V}\rangle]$ we can derive a fixed point $[\langle \mathbb{L}'; \mathbb{P}'; \mathbb{B}'; \mathbb{V}\rangle]$. It follows by Def. 8.5.6 that for any goal G, B, we have $\mathcal{A}^t_{[\![\mathcal{P}]\!]}([\![G, B]\!]_g) \subseteq [\![\mathcal{A}^!_{\mathcal{P}}(G, B)]\!]_g$.

Completeness: The **Introduce** and **Solve** rules of ω_t guarantee that for every $\sigma_t \in \Sigma_t$ there exists $\mathbb{T}, k$ such that

$$\mathcal{S}_!(\sigma_t) = [\langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V}\rangle] \quad \Rightarrow \quad \sigma_t \rightarrowtail_t^* \langle \emptyset; \mathbb{S}; \mathbb{B}; \mathbb{T}\rangle_k^{\mathbb{V}} \text{ s.t. } \mathrm{chr}(\mathbb{S}) = l(\mathbb{L}) \uplus p(\mathbb{P}) \tag{8.6}$$

With respect to **ApplyLinear**, assume $\sigma_t = \langle \emptyset; \mathbb{S}; \mathbb{B}; \mathbb{T}\rangle_k^{\mathbb{V}}$, $\sigma_! = \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V}\rangle$, $\tau_!$ such that $\mathrm{chr}(\mathbb{S}) = l(\mathbb{L}) \uplus p(\mathbb{P})$, and a rule $r \ @ \ H_1^l \uplus H_1^p \setminus H_2^l \uplus H_2^p \Leftrightarrow G \mid B_c, B_b$ such that $[\sigma_!] \rightarrowtail_!^r [\tau_!]$.

From $[\sigma_!] \rightarrowtail_!^r [\tau_!]$ follows that $\sigma_! \equiv_! \sigma'_! = \langle H_1^l \uplus H_2^l \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P}'; G \wedge \mathbb{B}'; \mathbb{V}\rangle$ and $\tau_! \equiv_! \tau'_! = \langle H_1^l \uplus B_c \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P}'; G \wedge \mathbb{B}' \wedge B_b; \mathbb{V}\rangle$, where $H_2^l \neq \emptyset$ and $\bar{y}, \bar{y}'$ are the local variables of $\sigma_!, \sigma'_!$. We assume w.l.o.g. that $\bar{y}, \bar{y}'$ are disjoint. Hence, Thm. 22 implies:

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{L} = H_1^l \uplus H_2^l \uplus \mathbb{L}') \wedge (\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}') \wedge \mathbb{B}' \wedge G))) \tag{8.7}$$

$$\mathcal{CT} \models \forall((\mathbb{B}' \wedge G) \rightarrow \exists \bar{y}.((\mathbb{L} = H_1^l \uplus H_2^l \uplus \mathbb{L}') \wedge (\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}') \wedge \mathbb{B}))) \tag{8.8}$$

By step 1 of our encoding, $[\![\mathcal{P}]\!]$ contains a rule

$$r' \ @ \ l(H_1^l) \uplus p(H_1^p) \uplus p(H_2^p) \setminus l(H_2^l) \Leftrightarrow G \mid l(B_c), B_b$$

We aptly decompose $\mathbb{L}$ into three components $\mathbb{L} = H_1^{l\prime} \uplus H_2^{l\prime} \uplus \mathbb{L}''$ such that:

$$\mathbb{L} = H_1^l \uplus H_2^l \uplus \mathbb{L}' \quad \Rightarrow \quad (H_1^{l\prime} = H_1^l) \wedge (H_2^{l\prime} = H_2^l) \wedge (\mathbb{L}'' = \mathbb{L}') \tag{8.9}$$

It is not guaranteed that $\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}' \Rightarrow (H_1^p \uplus H_2^p) \subseteq \mathbb{P}$. However, we can decompose $\mathbb{P}$ into two components $\mathbb{P} = H^{p\prime} \uplus \mathbb{P}''$ such that

$$\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}' \quad \Rightarrow \quad (H^{p\prime} \bowtie H_1^p \uplus H_2^p) \wedge (H^{p\prime} \subseteq H_1^p \uplus H_2^p) \tag{8.10}$$

Step 3 then guarantees that $[\![\mathcal{P}]\!]$ contains a rule

$$r'' \ @ \ l(H_1^l) \uplus p(H^p) \setminus l(H_2^l) \Leftrightarrow G' \wedge G \mid l(B_c), B_b$$

such that

$$\mathcal{CT} \models G' \leftrightarrow H^p \bowtie H_1^p \uplus H_2^p \tag{8.11}$$

and

$$\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}' \quad \Rightarrow \quad H^{p\prime} = H^p \tag{8.12}$$

Applying (8.12),(8.10), and (8.11) gives us

$$\mathcal{CT} \models (\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}') \rightarrow G' \tag{8.13}$$

Hence, from (8.7), we get

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{L} = H_1^l \uplus H_2^l \uplus \mathbb{L}') \wedge (\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}') \wedge G' \wedge \mathbb{B}' \wedge G)))$$

By Def. 1.1.5, we can thus derive $\sigma_t \rightarrowtail_t^{r''} \tau_t$ for

$$\tau_t = \langle l(B_c) \uplus B_b; \mathbb{S}'; \mathbb{B} \wedge (\tilde{\mathbb{B}} \wedge G' \wedge \mathbb{B}' \wedge G; \mathbb{T}'\rangle_k^{\mathbb{V}}$$

for some $\mathbb{T}'$ and where $\mathrm{chr}(\mathbb{S}') = l(H_1^{l\prime} \uplus \mathbb{L}'') \uplus p(H^{p\prime} \uplus \mathbb{P}'')$ and $\tilde{\mathbb{B}} = (\mathbb{L} = H_1^l \uplus H_2^l \uplus \mathbb{L}') \wedge (\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}')$. Consequently,

$$\mathcal{S}_!(\tau_t) = [\langle H_1^{l\prime} \uplus \mathbb{L}'' \uplus B_c; H^{p\prime} \uplus \mathbb{P}''; \mathbb{B} \wedge \tilde{\mathbb{B}} \wedge G' \wedge \mathbb{B}' \wedge G \wedge B_b; \mathbb{V}\rangle]$$

Applying (8.9) and Def. 8.3.3:1 gives us:

$$\mathcal{S}_!(\tau_t) = [\langle H_1^{l} \uplus \mathbb{L}' \uplus B_c; H^{p\prime} \uplus \mathbb{P}''; \mathbb{B} \wedge \tilde{\mathbb{B}} \wedge G' \wedge \mathbb{B}' \wedge G \wedge B_b; \mathbb{V}\rangle]$$

Since $\mathbb{P} = H^{p\prime} \uplus \mathbb{P}''$, we can apply the matching $(\mathbb{P} \bowtie H_1^p \uplus H_2^p \uplus \mathbb{P}')$ we find in the guard to get

$$\mathcal{S}_!(\tau_t) = [\langle H_1^{l} \uplus \mathbb{L}' \uplus B_c; H_1^{p} \uplus H_2^p \uplus \mathbb{P}'; \mathbb{B} \wedge \tilde{\mathbb{B}} \wedge G' \wedge \mathbb{B}' \wedge G \wedge B_b; \mathbb{V}\rangle]$$

As the variables in $\bar{y}, \bar{y}'$ are disjoint, we apply (8.8), (8.13), and Def. 8.3.3:2 to receive:

$$\mathcal{S}_!(\tau_t) = [\langle H_1^{l} \uplus \mathbb{L}' \uplus B_c; H_1^{p} \uplus H_2^p \uplus \mathbb{P}'; \mathbb{B}' \wedge G \wedge B_b; \mathbb{V}\rangle] = [\tau_!]$$

We proceed similarly for **ApplyPersistent**. Hence, for any $\sigma_t \in \Sigma_t, \tau_! \in \Sigma_!$ such that $\mathcal{S}_!(\sigma_t) \rightarrowtail_!^r [\tau_!]$, there exists a $\tau_t \in \Sigma_t$ s.t. $\sigma_t \rightarrowtail_t \tau_t$ and $\mathcal{S}_!(\tau_t') = [\tau_!]$.

Fixed points: Assume that $\sigma_! = \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V}\rangle$ is a fixed point in $\omega_!$. According to Def. 8.3.5, one of the following applies: (1) There is no rule $r \ @\ H_1^l \uplus H_1^p \setminus H_2^l \uplus H_2^p \Leftrightarrow G \mid B_c, B_b$ in $\mathcal{P}$ such that $\sigma_! \equiv_! \langle H_1^l \uplus H_1^p \uplus \mathbb{L}'; H_1^p \uplus H_2^p \uplus \mathbb{P}'; G \wedge \mathbb{B}'; \mathbb{V}\rangle$. (2) For every such rule that exists, its application violates the non-reflexivity condition, i.e. for a hypothetical follow-up state $[\tau_!]$, we have $[\sigma_!] = [\tau_!]$.

Now consider a state σ_t s.t. $\mathcal{S}_!(\sigma_t) = [\sigma_!]$. Hence, it is of the form $\sigma_t = \langle \emptyset; \mathbb{S}; \mathbb{B}; \mathbb{T}\rangle_k^V$ s.t. $\mathrm{chr}(\mathbb{S}) = l(\mathbb{L}) \uplus p(\mathbb{P})$. In case (1), no rules in $[\![\mathcal{P}]\!]$ are applicable to $\sigma_t = \langle \emptyset; l(\mathbb{L}) \uplus p(\mathbb{P}); \mathbb{B}; \mathbb{T}\rangle_k^V$, except for those of the form $c(\bar{t}) \setminus c(\bar{t}) \Leftrightarrow \top$. The program will quiesce in a state σ_t' s.t. $\mathcal{S}_!(\sigma_t') = [\sigma_!]$ after finitely many applications of such rules. In case (2) – assuming a non-pathological CHR program – all possible applications are of the type **ApplyPersistent** (cf. Lemma 8.5.9) . Consequently, all rules applicable to σ_t in $[\![\mathcal{P}]\!]$ are of the form $r' \ @\ l(H_1^l) \uplus p(\hat{H}^p) \Rightarrow G \mid p(B_c) \uplus B_b$ or $c(\bar{t}) \setminus c(\bar{t}) \Leftrightarrow \top$.

For each such rule $r' \ @\ l(H_1^l) \uplus p(\hat{H}^p) \Rightarrow G \mid p(B_c) \uplus B_b$, we can tell by $\sigma_! \equiv_! \tau_!$ that $p(B_c)$ is contained in $p(\mathbb{P})$ and B_b is contained in $\mathbb{B}$. Hence, we can apply r' to σ_t, followed by finitely many applications of rules of the form $c(\bar{t}) \setminus c(\bar{t}) \Leftrightarrow \top$ to finitely derive a state $\tau_t = \langle \emptyset; \mathbb{S}; \mathbb{B}'; \mathbb{T}'\rangle_{k'}^V$ such that $\mathcal{S}_!(\sigma_t) = \mathcal{S}_!(\tau_t)$ and r' is not applicable to τ_t. We repeat this for every applicable rule r'. After finitely many such sequences of derivation steps, no such rule remains applicable. Thus, we can finally derive a fixed point $\tau_t{}'$ such that $\mathcal{S}_!(\sigma_t) = \mathcal{S}_!(\tau_t{}')$.

It follows by Def. 8.5.6 that for any goal G, B, we have $[\![\mathcal{A}^!_{\mathcal{P}}(G, B)]\!]_g \subseteq \mathcal{A}^t_{[\![\mathcal{P}]\!]}([\![G, B]\!]_g)$. □

Example 8.5.16 (Termination Correspondence)**.** *The termination behavior of $\omega_!$ programs encoded in ω_t, via the encoding used to prove Lemma 8.5.15, changes. Consider a program $\mathcal{P}$ consisting only of the rule $a \Longrightarrow a$ that is clearly terminating in $\omega_!$. Its corresponding encoded program $[\![\mathcal{P}]\!]$ is given below.*

$$\begin{array}{lllll} r_1 \ @ & a(l) & \Longrightarrow & a(p) \\ r_2 \ @ & a(p) & \Longrightarrow & a(p) \\ r_3 \ @ & a(p) \setminus a(p) & \Leftrightarrow & \top \end{array}$$

It is an acceptable encoding according to Definition 8.5.7, and hence, answers are preserved. Nevertheless, there exists the following infinite computation.

$$
\begin{array}{ll}
\sigma = & \langle a(l); \emptyset; \top; \emptyset \rangle_0^{\emptyset} \\
\rightarrowtail_t & \langle \emptyset; a(l)\#0; \top; \emptyset \rangle_1^{\emptyset} \\
\rightarrowtail_t^{r_1} & \langle a(p); a(l)\#0; \top; (r_1, 0) \rangle_1^{\emptyset} \\
\rightarrowtail_t & \langle \emptyset; a(l)\#0, a(p)\#1; \top; (r_1, 0) \rangle_2^{\emptyset} \\
\rightarrowtail_t^{r_2} & \langle a(p); a(l)\#0, a(p)\#1; \top; (r_1, 0), (r_2, 1) \rangle_2^{\emptyset} \\
\rightarrowtail_t & \langle \emptyset; a(l)\#0, a(p)\#1, a(p)\#2; \top; (r_1, 0), (r_2, 1) \rangle_3^{\emptyset} \\
\rightarrowtail_t^{r_2} & \langle a(p); a(l)\#0, a(p)\#1, a(p)\#2; \top; (r_1, 0), (r_2, 1), (r_2, 2) \rangle_3^{\emptyset} \\
\rightarrowtail_t & \ldots
\end{array}
$$

The reason for this difference is found in rules r_2 and r_3: they enforce set semantics on the constraints, supposedly corresponding to irreflexivity in $\omega_!$. However, the non-determinism of ω_t seems to hinder proper enforcing of irreflexivity via rules.

Lemma 8.5.17 ($\omega_p \not\twoheadrightarrow \omega_!$)**.** *There exists no acceptable encoding of ω_p into $\omega_!$.*

Proof. Follows from [Gabbrielli et al., 2009]. Gabbrielli et al. [2009] consider only data sufficient answers, however, as there exists no acceptable encoding of the program given in their proof, the negative result carries over to the generic case of answers. □

Lemma 8.5.18 ($\omega_! \rightarrow \omega_p$)**.** *There exists an acceptable encoding of $\omega_!$ into ω_p.*

Proof. We show how to encode any $\omega_!$ program $\mathcal{P}$ in ω_p. For every n-ary constraint c/n in $\mathcal{P}$, there exists a constraint $c/(n+1)$ in $[\![\mathcal{P}]\!]$. In the following, for a multiset of user-defined $\omega_!$-constraints $M = \{c_1(\bar{t}_1), \ldots, c_n(\bar{t}_n)\}$ let $l(M) ::= \{c_1(l, \bar{t}_1), \ldots, c_n(l, \bar{t}_n)\}$, $p(M) ::= \{c_1(p, \bar{t}_1), \ldots, c_n(p, \bar{t}_n)\}$, and $c(M) ::= \{c_1(c, \bar{t}_1), \ldots, c_n(c, \bar{t}_n)\}$. The encoded program $[\![\mathcal{P}]\!]$ is constructed as follows:

Apply rules 1-3 from the proof of Lemma 8.5.15, but in rule 2 replace $p(B_c)$ with $c(B_c)$. Assign to each of these rules the constant priority 3. Additionally, add the following rules to $[\![\mathcal{P}]\!]$ for each constraint c/n where $\bar{t}$ is a sequence of n different variables:

$$
\begin{array}{l}
1 :: c(p, \bar{t}) \backslash c(c, \bar{t}) \Leftrightarrow \top \\
2 :: c(c, \bar{t}) \Leftrightarrow c(p, \bar{t})
\end{array}
$$

The translation of goals is defined as $[\![\mathbb{L} \uplus ! \mathbb{P}]\!]_g ::= l(\mathbb{L}) \uplus p(\mathbb{P})$.
Soundness: Let $\mathcal{S}_! : \Sigma_p \rightarrow \Sigma_!$, $\sigma_p = \langle l(\mathbb{L}) \uplus p(\mathbb{P}) \uplus c(\mathbb{P}_c) \uplus \mathbb{B}'; \mathbb{S}; \mathbb{B}; \mathbb{T} \rangle_k^{\mathbb{V}} \mapsto \langle \mathbb{L} \uplus \mathbb{L}'; \mathbb{P} \uplus \mathbb{P}' \uplus \mathbb{P}_c \uplus \mathbb{P}'_c; \mathbb{B} \wedge \mathbb{B}'; \mathbb{V} \rangle$ where $\mathrm{chr}(\mathbb{S}) = l(\mathbb{L}') \uplus p(\mathbb{P}') \uplus c(\mathbb{P}'_c)$. In the following, we will show that for all $\sigma_p, \tau_p \in \Sigma_p$, $\sigma_p \rightarrowtail_p^* \tau_p$ implies $\mathcal{S}_!(\sigma_p) \rightarrowtail_!^* \mathcal{S}_!(\tau_p)$.

The proof is analogous to Lemma 8.5.15 for the rules of priority 3. As $c(\bar{t}) \bowtie c(\bar{t}) \uplus c(\bar{t})$ rules of priority 1 and 2 are invariant to $\mathcal{S}_!$.

Now assume τ_p is a *fixed point* w.r.t. $[\![\mathcal{P}]\!]$. Analogously to Lemma 8.5.15, $\mathcal{S}_!(\tau_p)$ is a fixed point w.r.t. $\mathcal{P}$. The only difference being $c(B_c)$ in the body instead of $p(B_c)$, but rules of priority 1 and 2 would then be applicable to convert $c(B_c)$ into $p(B_c)$ modulo set semantics. Therefore, it follows that $\mathcal{A}^p_{[\![\mathcal{P}]\!]}([\![G]\!]_g) \subseteq [\![\mathcal{A}^!_{\mathcal{P}}(G)]\!]_g$.
Completeness: Analogously to Lemma 8.5.15, we have that for any $\sigma_p \in \Sigma_p$, $\tau_! \in \Sigma_!$ such that $\mathcal{S}_!(\sigma_p) \rightarrowtail_!^r [\tau_!]$, there exists a $\tau_p \in \Sigma_p$ such that $\sigma_p \rightarrowtail_p^* \tau_p$ and $\mathcal{S}_!(\tau_p) = [\tau_!]$. The only change to the proof is that after applying a rule of the encoded program we also apply all possible **Introduce** and **Solve** transitions, as well as all rule applications with priorities 1 and 2 (all these operations are invariant to $\mathcal{S}_!$). Hence, the resulting state τ_p

contains only identified constraints whose first argument is either l or p. All constraints with argument c are either replaced by the corresponding one with argument p by the rule of priority 2, or they are removed, because a corresponding constraint already exists.

Now assume $\sigma_! = \langle \mathbb{L}; \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle$ is a *fixed point* of $\mathcal{P}$, then there exists $\sigma_p \in \Sigma_p$ with $\mathcal{S}_!(\sigma_p) = [\sigma_!]$. There are two possible cases:

1. $[\sigma_!]$ is not applicable to any rule $r \in \mathcal{P}$ (when disregarding irreflexivity)

2. all rule applications would violate irreflexivity

In case 1, σ_p clearly is a fixed point as well (otherwise the above soundness result violates the assumption).

Therefore, consider case 2. We assume non-pathological programs, so that, according to Lemma 8.5.9, **ApplyLinear** never violates irreflexivity. Hence, there exists a rule in $[\![\mathcal{P}]\!]$:

$$3 :: r' \ @ \ l(H^l) \uplus p(H^p) \Longrightarrow G \mid c(B_c), B_b$$

In the following, for a set M of constraints let $\#M$ denote the corresponding set of identified constraints. Assume σ_p is no fixed point, then $\sigma_p = \langle \emptyset; \#l(\hat{H}^l) \cup \#p(\hat{H}^p) \cup \mathbb{S}; \mathbb{B}; \mathbb{T} \rangle_k^{\mathbb{V}}$ and $\mathcal{CT} \models \forall(\mathbb{B} \rightarrow (A \wedge G))$ with $\sigma_p \rightarrowtail_p^{r'} \tau_p = \langle c(B_c) \uplus B_b; \#l(\hat{H}^l) \cup \#p(\hat{H}^p) \cup \mathbb{S}; \mathbb{B} \wedge A; \mathbb{T}' \rangle_k^{\mathbb{V}}$, where $A ::= \mathrm{chr}(\#l(\hat{H}^l)) = l(H^l) \wedge \mathrm{chr}(\#p(\hat{H}^p)) = p(H^p)$. Applying **Introduce** and **Solve**, we get $\tau_p \rightarrowtail_p^* \langle \emptyset; \#l(\hat{H}^l) \cup \#p(\hat{H}^p) \cup \mathbb{S} \cup \#c(B_c); \mathbb{B} \wedge B_b \wedge A; \mathbb{T}' \rangle_m^{\mathbb{V}}$.

The rule r' corresponds to a rule r in $\mathcal{P}$ and $\sigma_!$ is applicable to r, except for irreflexivity (this follows from soundness). The irreflexivity and Theorem 22 imply $\mathcal{CT} \models \mathbb{B} \rightarrow \exists \bar{x}.(H^p \bowtie H^p \uplus B_c) \wedge \mathbb{B} \wedge B_b$. Therefore, $\mathcal{CT} \models (\mathbb{B} \wedge B_b \wedge A) \rightarrow (\hat{H}^p \bowtie \hat{H}^p \uplus B_c)$. It follows that $\mathcal{CT} \models (\mathbb{B} \wedge B_b \wedge A) \rightarrow \forall c(c, \bar{t}) \in c(B_c). \exists c(p, \bar{t}') \in p(\hat{H}^p). \bar{t} = \bar{t}'$.

Therefore, for each $c(c, \bar{t}) \in c(B_c)$ we can apply the corresponding rule

$$1 :: c(p, \hat{t}) \backslash c(c, \hat{t}) \Leftrightarrow \top,$$

as $\mathcal{CT} \models \forall(\mathbb{B} \wedge B_b \wedge A \rightarrow \exists \bar{x}.(\mathrm{chr}(c(p, \bar{t}')) = c(p, \hat{t}) \wedge \mathrm{chr}(c(c, \bar{t})) = c(c, \hat{t})))$. Therefore, each constraint in $c(B_c)$ is removed by rules of priority 1 and we get $\sigma_p \rightarrowtail_p^* \langle \emptyset; \#l(\hat{H}^l) \cup \#p(\hat{H}^p) \cup \mathbb{S}; \mathbb{B}; \mathbb{T}' \rangle_m^{\mathbb{V}} = \tau_p'$, such that the above rule application is prohibited by $\mathbb{T}'$.

Hence, we can w.l.o.g. choose τ_p' as σ_p above and repeat the argument. Therefore, we get a state in which the token store prohibits firing any more propagation rules. As no other rules are applicable either, this state is a fixed point corresponding to $\sigma_!$ as well. □

Lemma 8.5.19 ($\omega_! \not\rightarrow \omega_e$)**.** *There exists no acceptable encoding of $\omega_!$ into ω_e.*

Proof. Consider the $\omega_!$ program $\mathcal{P} = (a \Longrightarrow b)$. Since $\mathcal{A}^!_{\mathcal{P}}(a) = \{[\langle a, !b; \top; \emptyset \rangle]\}$, an acceptable encoding has to satisfy $\mathcal{A}^e_{[\![\mathcal{P}]\!]}([\![a]\!]_g) = \{[\langle [\![a, !b]\!]_g; \top; \emptyset \rangle]\} = \{[\langle [\![a]\!]_g \uplus [\![!b]\!]_g; \top; \emptyset \rangle]\}$. Therefore, $[\langle [\![a]\!]_g; \top; \emptyset \rangle] \rightarrowtail_e^+ [\langle [\![a]\!]_g \wedge [\![!b]\!]_g; \top; \emptyset \rangle]$ where the result state has to be a final state, which is a contradiction to monotonicity of ω_e. □

Lemma 8.5.20 ($\omega_e \rightarrow \omega_!$)**.** *There exists an acceptable encoding of ω_e into $\omega_!$.*

Proof (sketch). Replace propagation rules with simplification rules that contain a copy of the head in their bodies. □

8.5.5 Implementation

In this section, we discuss the implementation of a CHR system that adheres to the operational semantics $\omega_!$. Currently, no direct implementation exists, so that we only discuss how it would differ from existing implementations. Afterwards, we present a source-to-source transformation that allows us to simulate $\omega_!$ execution in ω_p. The existing implementation of ω_p can therefore be used for performing derivations following the operational semantics $\omega_!$.

Correspondence to Existing Implementations

The overall behavior of $\omega_!$ is not significantly different from ω_t and ω_r. Clearly, $\omega_!$ is formally a non-deterministic system like ω_t, so that a refinement similar to ω_r may be considered. However, we consider it an allowed freedom of implementations to perform a fixed rule and constraint selection. Therefore, even without formally refining $\omega_!$, an implementation may, for example, try rule applications in textual order.

The treatment of built-in and CHR constraints remains almost unchanged. Persistent constraints can be implemented just like linear CHR constraints, when labeling them as persistent. Additionally, an implementation should detect duplicates of persistent constraints and eliminate them.

Slightly more effort is required to adjust the code responsible for matching CHR constraints to rule heads. Firstly, while linear CHR constraints can only be used to match one head constraint, $\omega_!$ requires that a persistent constraint can match multiple head constraints. Secondly, a case distinction is required to distinguish between **ApplyLinear** and **ApplyPersistent** rule applications, as the resulting body constraints have to be inserted either linearly or persistently.

Finally, the main difference to existing implementations is the irreflexive transition system used for $\omega_!$. However, in terms of an implementation, the required effort is reduced by the following insight: If the post-transition CHR state is considered as a rule, with all CHR constraints in its head and the conjunction of built-ins as guard, one can simply try to match the pre-transition state to that rule. The necessary code can be reused from the normal matching required for rule applications. If there is such a match, then the CHR constraints are equivalent and the pre-transition built-in store already implies the post-transition built-in store, hence, irreflexivity would be violated.

Additional effort needs to be invested to ensure that persistent constraints are equivalent in the pre- and post-transition states. For example, the rule $a \Longrightarrow a$ turns a linear a-constraint into a persistent a-constraint, but the distinction between these two is lost, when only considering an a-constraint in a rule head. So a minor adjustment to the matching code is required to ensure that persistent and linear constraints match each other, when testing the irreflexivity condition.

Implementation via Source-to-Source Transformation

In this section, we provide an implementation of the operational semantics $\omega_!$ in the form of a source-to-source transformation. A CHR program $\mathcal{P}$ is transformed into a program $[\![\mathcal{P}]\!]$ such that $[\![\mathcal{P}]\!]$'s execution in ω_p is sound and complete with respect to the execution of $\mathcal{P}$ in $\omega_!$. This transformation is based on Lemma 8.5.18 and assumes a CHR program $\mathcal{P}$ without pathological rules.

For every n-ary constraint c/n in $\mathcal{P}$, there exists a constraint $c/(n+1)$ in $[\![\mathcal{P}]\!]$. In the following, for a multiset of user-defined $\omega_!$-constraints $M = \{c_1(\bar{t}_1), \ldots, c_n(\bar{t}_n)\}$ let

- $l(M) ::= \{c_1(l, \bar{t}_1), \ldots, c_n(l, \bar{t}_n)\}$,
- $p(M) ::= \{c_1(p, \bar{t}_1), \ldots, c_n(p, \bar{t}_n)\}$, and
- $c(M) ::= \{c_1(c, \bar{t}_1), \ldots, c_n(c, \bar{t}_n)\}$.

The rules of $[\![\mathcal{P}]\!]$ are constructed via the following source-to-source transformation.

1. For every rule $r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B$ in $\mathcal{P}$, and all multisets $H_1^l, H_1^p, H_2^l, H_2^p$ s.t. $H_1^l \uplus H_1^p = H_1$ and $H_2^l \uplus H_2^p = H_2$ and $H_2^l \neq \emptyset$, the following rule is in $[\![\mathcal{P}]\!]$:

$$3 :: l(H_1^l) \uplus p(H_1^p) \uplus p(H_2^p) \setminus l(H_2^l) \Leftrightarrow G \mid l(B_c), B_b$$

2. For every rule $r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b$ in $\mathcal{P}$, and all multisets H_1^l, H_1^p s.t. $H_1^l \uplus H_1^p = H_1$, the following rule is in $[\![\mathcal{P}]\!]$:

$$3 :: l(H_1^l) \uplus p(H_1^p) \uplus p(H_2) \Longrightarrow G \mid c(B_c), B_b$$

3. For every rule $\{c(p, \bar{t}), c(p, \bar{t}')\} \uplus H_1 \setminus H_2 \Leftrightarrow G \mid B$ in $[\![\mathcal{P}]\!]$, add also the following rule:

$$3 :: \{c(p, \bar{t})\} \uplus H_1 \setminus H_2 \Leftrightarrow \bar{t} = \bar{t}' \wedge G \mid B$$

4. For every user-defined constraint c/n in $\mathcal{P}$, add the following rules, where $\bar{t}$ is a sequence of n different variables:

$$\begin{array}{l} 1 :: c(p, \bar{t}) \backslash c(c, \bar{t}) \Leftrightarrow \top \\ 2 :: c(c, \bar{t}) \Leftrightarrow c(p, \bar{t}) \end{array}$$

Example 8.5.21 (Encoding of Transitive Hull)**.** *We consider the transitive hull program from Example 8.3.1:*

$$t \quad @ \quad e(X, Y), e(Y, Z) \quad \Longrightarrow \quad e(X, Z)$$

According to the encoding given above, the program is transformed as follows:

$$\begin{array}{lrcl}
3 & :: & e(l, X, Y), e(l, Y, Z) & \Longrightarrow \quad e(c, X, Z) \\
3 & :: & e(l, X, Y), e(p, Y, Z) & \Longrightarrow \quad e(c, X, Z) \\
3 & :: & e(p, X, Y), e(l, Y, Z) & \Longrightarrow \quad e(c, X, Z) \\
3 & :: & e(p, X, Y), e(p, Y, Z) & \Longrightarrow \quad e(c, X, Z) \\
\\
3 & :: & e(p, X, Y) & \Longrightarrow \quad X = Y \wedge Y = Z \mid e(c, X, Z) \\
\\
1 & :: & e(p, X, Y) \backslash e(c, X, Y) & \Leftrightarrow \quad \top \\
2 & :: & e(c, X, Y) & \Leftrightarrow \quad e(p, X, Y)
\end{array}$$

The grouping of the rules above reflects the transformation steps 2, 3, and 4. Transformation step 1 is not productive in this example. The fifth rule above is operationally equivalent to $3 :: e(p, X, X) \Longrightarrow e(c, X, X)$*, and hence, is redundant, as the resulting constraint will immediately be removed again by the rule with priority 1. Furthermore, transformation step 3 also adds an additional symmetric version of the fifth rule, which was omitted here, as it is operationally equivalent as well.*

Execution of a transformed program in ω_p is equivalent to execution of the original program in $\omega_!$, as the above is the acceptable encoding used in the proof of Lemma 8.5.18. As opposed to the acceptable encoding into ω_t, this encoding also preserves fixed points, which makes it suitable for our implementation via source-to-source transformation.

Example 8.5.22 (Example Runs of ω_p and $\omega_!$ Programs)**.** *The following example derivation shows how the translated program terminates with a state that corresponds to the result of an execution of the original program in $\omega_!$. For clarity's and brevity's sake, we do not show all intermediate states and we do not give the states' respective token stores explicitly.*

$$
\begin{array}{ll}
 & \langle e(l,A,B), e(l,B,A); \emptyset; \top; \emptyset\rangle_0^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1; \top; \emptyset\rangle_2^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(c,A,A)\#2; \top; \ldots\rangle_3^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3; \top; \ldots)\rangle_4^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3, e(c,B,B)\#4; \top; \ldots\rangle_5^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3, e(p,B,B)\#5; \top; \ldots)\rangle_6^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3, e(p,B,B)\#5, e(c,A,B)\#6; \top; \ldots\rangle_7^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3, e(p,B,B)\#5, e(p,A,B)\#7; \top; \ldots)\rangle_8^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3, e(p,B,B)\#5, e(p,A,B)\#7, e(c,B,A)\#8; \top; \ldots\rangle_9^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3, e(p,B,B)\#5, e(p,A,B)\#7, e(p,B,A)\#9; \top; \ldots)\rangle_{10}^{\{A,B\}} \\
\rightarrowtail_p^* & \langle \emptyset; e(l,A,B)\#0, e(l,B,A)\#1, e(p,A,A)\#3, e(p,B,B)\#5, e(p,A,B)\#7, e(p,B,A)\#9; \top; \ldots)\rangle_{24}^{\{A,B\}} \\
\not\rightarrowtail_p &
\end{array}
$$

The above computation corresponds to the following execution in $\omega_!$:

$$
\begin{array}{ll}
[\sigma] = & [\langle e(A,B), e(B,A); \emptyset; \top; \{A,B\}\rangle] \\
\rightarrowtail_!^t & [\langle e(A,B), e(B,A); e(A,A); \top; \{A,B\}\rangle] \\
\rightarrowtail_!^t & [\langle e(A,B), e(B,A); e(A,A), e(B,B); \top; \{A,B\}\rangle] \\
\rightarrowtail_!^t & [\langle e(A,B), e(B,A); e(A,A), e(B,B), e(A,B); \top; \{A,B\}\rangle] \\
\rightarrowtail_!^t & [\langle e(A,B), e(B,A); e(A,A), e(B,B), e(A,B), e(B,A); \top; \{A,B\}\rangle] \\
\not\rightarrowtail_! &
\end{array}
$$

The above example demonstrates that a direct implementation of $\omega_!$ performs less computation steps than its simulation via ω_p. Therefore, the source-to-source transformation is important in that it allows us to directly experiment with $\omega_!$ programs, but ultimately, a direct implementation is more desirable.

8.6 Related and Future Work

We have already discussed different available operational semantics for CHR in Chapter 1 and seen that they tend to be either analytical or pragmatic, i.e. implementation-oriented. Our operational semantics $\omega_!$ fills this gap, as it has a strong declarative foundation and is implementable in a terminating fashion.

In particular, all operational semantics that are based on the token-store approach to deal with the trivial non-termination problem (cf. for example [Abdennadher, 1997, Duck et al., 2004]), suffer from incompleteness. In contrast, our approach based on persistent constraint offers a more natural way that better corresponds to the abstract operational semantics ω_{va}.

The set-based operational semantics ω_{set} given by Sarna-Starosta and Ramakrishnan [2007], on the other hand, avoids the token-store in favor of a low-level change in the implementation. From an analytical point of view this change is undesirable, because it leads to an undetermined number of propagation rule applications. In ω_{set}, execution is similar to ω_r, in that an active constraint is selected and has to find partner constraints for rule applications. Once such a constraint finds no more possible rule applications, it

becomes inactive, but can later be awakened again due to changes in the built-in store. This leads to a reactivation, which in turn may lead to another firing of a propagation rule. This overall process is hard to deal with in a declarative way, because it requires a detailed observation of the runtime behavior.

Sneyers et al. [2005] already showed that CHR is Turing-complete, even for various subclasses of the language [Sneyers, 2008, Gabbrielli et al., 2010]. Hence, our operational semantics $\omega_!$ is Turing-complete as well, due to it being sound and complete with respect to ω_{va}. In order to compare it with the existing operational semantics, we continued along the lines of Gabbrielli et al. [2009] by showing its relative expressivity with respect to ω_{va}, ω_t, and ω_p. This also yielded a source-to-source transformation that can be used to implement $\omega_!$ via ω_p. For the future, a direct implementation of $\omega_!$ would be preferable though.

The current formulations of $\omega_!$ and ω_e promise to be beneficial in future CHR research. In [Raiser, 2010], we demonstrate its viability and wherever previous work exists, we can see that ω_e allows for more succinct and clear formulations. Our axiomatic definition of state equivalence from [Raiser et al., 2009] has already been reused in [Sneyers et al., 2010].

For the operational semantics $\omega_!$ it might be possible to include the following additional axiom for state equivalence

$$\langle L \uplus \mathbb{L}; L \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle \equiv_! \langle \mathbb{L}; L \uplus \mathbb{P}; \mathbb{B}; \mathbb{V} \rangle.$$

Intuitively, this means that a linear constraint is superseded by its persistent variant. We initially decided against this axiom, because it widens the gap to linear-logic, in which a persistent constraint is not equivalent to itself and a linear copy. However, if one accepts this limitation, the above axiom would help to simplify the formulation of the operational semantics significantly. Given this axiom, we can for example ensure that each **ApplyLinear** transition involves exclusively linear constraints. However, we cannot simply include this axiom, but need to reevaluate our above proofs, especially soundness and completeness of the resulting operational semantics.

Another advantage of such an axiom would be more intuitive results for confluence. Consider the following rule under the current definition of $\omega_!$.

$$a, b \Leftrightarrow c$$

Given the initial state $\langle a, b; a; \top; \emptyset \rangle$ we can apply the rule in two different ways, either with the linear or persistent copy of the a-constraint. This results in the following two states, which are not equivalent without the above axiom, but would be equivalent if it was included.

$$\langle c; a; \top; \emptyset \rangle \not\equiv_! \langle a, c; a; \top; \emptyset \rangle$$

In our algebraic investigation of the merge operator we found it to form a commutative monoid together with the set of equivalence classes of states. This was just a first foray into this line of research, namely the algebraization of CHR. We believe it is possible to profit from the abundance of results available in this field, when further investigating the algebraic structures underlying CHR.

For example, the CHR transition system might be expressible as a category, allowing us to investigate category theoretical constructions for CHR, like pushouts or pullbacks. Similarly, the previously mentioned commutative monoid can be extended to an abelian group, for example via the Grothendieck group construction. This would add inverses of states, or in other words, gives us the algebraic option of subtracting from states. This

could provide a formalism that allows us to undo operations, like binding of variables, and hence, extend CHR to a non-committed-choice language, similar to CHR$^\vee$.

On a more pragmatic note, associativity of the merge operator makes it a suitable formalism for investigating parallelism in CHR. If a part of a CHR state allows a computation while being independent from the rest of the state, we can express this as $[\sigma] = [\sigma_0] \diamond [\sigma']$. Applying this idea to all independent parallel computations possible in a state, then yields

$$[\sigma] = [\sigma_0] \diamond [\sigma_1] \diamond \ldots \diamond [\sigma_n] \diamond [\sigma'],$$

which clearly separates n independent parallel computations.

Bibliography

Slim Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In G. Smolka, editor, *CP '97*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, November 1997.

Slim Abdennadher. *Rule-based Constraint Programming: Theory and Practice*. Habilitationsschrift, Institute of Computer Science, LMU, Munich, Germany, July 2001.

Slim Abdennadher and Thom Frühwirth. Operational Equivalence of CHR Programs and Constraints. In J. Jaffar, editor, *CP '99*, volume 1713 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, October 1999.

Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints*, 4(2):133–165, 1999.

Hariolf Betz and Thom Frühwirth. A Linear-Logic Semantics for Constraint Handling Rules. In P. van Beek, editor, *CP '05*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, October 2005.

Hariolf Betz, Frank Raiser, and Thom Frühwirth. Persistent Constraints in Constraint Handling Rules. In U. Geske and A. Wolf, editors, *WLP '09*, pages 155–166. Universittsverlag Potsdam, September 2009.

Hariolf Betz, Frank Raiser, and Thom Frühwirth. A Complete and Terminating Execution Model for Constraint Handling Rules. *Theory and Practice of Logic Programming*, 10 (4–6):597–610, 2010.

Gregory J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Melbourne, Australia, December 2005.

Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer-Verlag, September 2004.

Gregory J. Duck, Peter J. Stuckey, and Martin Sulzmann. Observable Confluence for Constraint Handling Rules. In T. Schrijvers and T. Frühwirth, editors, *CHR '06*, pages 61–76. K.U.Leuven, Department of Computer Science, Technical report CW 452, July 2006.

Gregory J. Duck, Peter J. Stuckey, and Martin Sulzmann. Observable Confluence for Constraint Handling Rules. In V. Dahl and I. Niemelä, editors, *ICLP '07*, volume 4670 of *Lecture Notes in Computer Science*, pages 224–239. Springer-Verlag, September 2007.

Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.

Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009. ISBN 0-521-87776-8.

Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, 2003. ISBN 3-540-67623-6.

Thom Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic Constraint Handling Rules. *Electronic Notes in Theoretical Computer Science*, 76:1–16, 2002.

Maurizio Gabbrielli, Jacopo Mauro, and Maria Chiara Meo. On the Expressive Power of Priorities in CHR. In A. Porto and F. J. López-Fraguas, editors, *PPDP '09*, pages 267–276. ACM Press, September 2009.

Maurizio Gabbrielli, Jacopo Mauro, Maria Chiara Meo, and Jon Sneyers. Decidability Properties for Fragments of CHR. *Theory and Practice of Logic Programming*, 10(4-6): 611–626, 2010.

Rémy Haemmerlé and François Fages. Abstract Critical Pairs and Confluence of Arbitrary Binary Relations. In F. Baader, editor, *RTA '07*, volume 4533 of *Lecture Notes in Computer Science*, pages 214–228. Springer-Verlag, June 2007.

Johannes Langbein, Frank Raiser, and Thom Frühwirth. A State Equivalence and Confluence Checker for CHR. In P. Van Weert and L. De Koninck, editors, *CHR '10*, pages 1–8. K.U.Leuven, Department of Computer Science, Technical report CW 588, July 2010.

Marc Meister. *Advances in Constraint Handling Rules*. PhD thesis, Ulm University, Ulm, Germany, 2008.

Paolo Pilozzi and Danny De Schreye. Proving Termination by Invariance Relations. In P. M. Hill and D. S. Warren, editors, *ICLP '09*, volume 5649 of *Lecture Notes in Computer Science*, pages 499–503. Springer-Verlag, July 2009.

Frank Raiser. *Graph Transformation Systems in Constraint Handling Rules: Improved Methods for Program Analysis*. PhD thesis, Ulm University, November 2010.

Frank Raiser and Thom Frühwirth. Strong Joinability Analysis for Graph Transformation Systems in CHR. *Electronic Notes in Theoretical Computer Science – TERMGRAPH '09: Proc. 5th Intl. Workshop Computing with Terms and Graphs*, 253(4): 91–111, 2009.

Frank Raiser and Paolo Tacchella. On Confluence of Non-Terminating CHR Programs. In K. Djelloul, G. J. Duck, and M. Sulzmann, editors, *CHR '07*, pages 63–76, September 2007.

Frank Raiser, Hariolf Betz, and Thom Frühwirth. Equivalence of CHR States Revisited. In F. Raiser and J. Sneyers, editors, *CHR '09*, pages 34–48. K.U.Leuven, Department of Computer Science, Technical report CW 555, July 2009.

Beata Sarna-Starosta and C.R. Ramakrishnan. Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In M. Hanus, editor, *PADL '07*, volume 4354 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag, January 2007.

Ehud Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

Robert J. Simmons and Frank Pfenning. Linear Logical Algorithms. In L. Aceto, I. Damgård, L. Ann Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP '08*, volume 5126 of *Lecture Notes in Computer Science*, pages 336–347. Springer-Verlag, July 2008.

Jon Sneyers. *Optimizing Compilation and Computational Complexity of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, November 2008.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. The Computational Power and Complexity of Constraint Handling Rules. In T. Schrijvers and T. Frühwirth, editors, *CHR '05*, pages 3–17. K.U.Leuven, Department of Computer Science, Technical report CW 421, October 2005.

Jon Sneyers, Tom Schrijvers, and Bart Demoen. The Computational Power and Complexity of Constraint Handling Rules. *ACM Transactions on Programming Languages and Systems*, 31(2):1–42, 2009.

Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based Probabilistic Logic Learning. *Theory and Practice of Logic Programming*, 10(4-6):433–447, 2010.

Chapter 9

Abstract Interpretation

Author:	Tom Schrijvers
Thesis Title:	Analyses, Optimizations and Extensions of Constraint Handling Rules
School:	K.U.Leuven, Belgium
Publication Year:	2005

Foreword

Optimized compilation is made up of two parts: 1) a program analysis, and 2) a program transformation. The analysis identifies the opportunity or validity of the optimization and the transformation applies it. Generally, both are described in an adhoc fashion in the CHR literature. This chapter provides a more principled approach to program analysis for CHR. Abstract interpretation [Cousot and Cousot, 1977] is a generic technique to derive a program analysis framework from a language's operational semantics. This chapter assumes the reader is familiar with the basics of abstract interpretation.

While the chapter at hand focuses on the refined operational semantics, abstract interpretation may well be the most straightforward approach to transfer existing and develop new program analyses for the operational semantics that have arisen in recent years. However, an abstract interpretation framework does require an initial development cost, which makes it less suitable for systems in their initial stages. To date only Duck's HAL CHR system and the K.U.Leuven CHR system incorporate abstract interpretation.

9.1 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 et al., 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 et al., 2004b] 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 9.2 presents the refined denotational semantics of CHR that will be abstractly interpreted. The general abstract interpretation framework is then defined in Section 9.3. Two instances of the framework, late storage analysis and groundness analysis, illustrate the framework in Sections 9.4 and 9.5 respectively. The implementation and experimental evaluation of these analyses are reported on in Section 9.6. We conclude in Section 9.7.

9.2 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 [Duck et al., 2004b] 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 et al., 1994]).

It is shown in [Duck et al., 2004a] 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 9.1 The numbered version of a `gcd` program, similar to Example 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 9.2.1 and 9.2.2 we present the execution state and semantic function of ω_d. Section 9.2.3 illustrates the semantics on an example.

9.2.1 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 9.2.2) 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 [Schrijvers, 2005a, Section 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.

9.2.2 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:

$$\mathcal{S} : \mathbf{Prog} \to (\mathbf{\Sigma} \hookrightarrow \mathbf{\Sigma})$$

The definition of the semantic function is given in Table 9.1. 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** case 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.

9.2.3 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.
```

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.

$$
\begin{array}{lll}
\mathcal{S}[\![\mathcal{P}]\!](\langle p, [], \emptyset, \mathit{true}, \emptyset\rangle_1) & & (\textbf{Activate}) \\
\quad = & \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1 : 1, [], \{p\#1\}, \mathit{true}, \bullet\rangle_2) & (\textbf{Propagate}) \\
\quad = & \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1 : 2, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) & (\neg\textbf{Simplify}) \\
\quad = & \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1 : 3, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) & (\neg\textbf{Propagate}) \\
\quad = & \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1 : 4, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) & (\textbf{Propagate}) \\
\quad = & \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1 : 5, [], \{q\#2\}, \mathit{true}, \bullet\rangle_4) & (\neg\textbf{Simplify}) \\
\quad = & \mathcal{S}[\![\mathcal{P}]\!](\langle p\#1 : 6, [], \{q\#2\}, \mathit{true}, \bullet\rangle_4) & (\textbf{Drop}) \\
\quad = & \langle \Box, [], \{q\#2\}, \mathit{true}, \bullet\rangle_4 &
\end{array}
$$

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 9.1: 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

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \textit{simplify-condition} \\ \qquad \text{then} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), A, H_1 \uplus S', \theta \wedge B, T \cup \{t\}\rangle_n) \\ \qquad \text{else} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j+1, A, S, B, T\rangle_n)\end{array}$$

where *simplify-condition* is that there exists a matching substitution θ such that

$$\left\{\begin{array}{l} S = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S' \\ c = \theta(d) \\ chr(H_1) = \theta(H_1') \wedge chr(H_2) = \theta(H_2') \wedge chr(H_3) = \theta(H_3') \\ \mathcal{D}_b \models B \rightarrow \bar{\exists}_r(\theta \wedge g) \\ t = id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{array}\right.$$

---

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

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B_k \\ \qquad \text{then} \ \langle \Box, A, S_k, B_k, T_k\rangle_{n_k} \\ \qquad \text{else} \ \ \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j+1, A, S_k, B_k, T_k\rangle_{n_k})\end{array}$$

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:

$$\begin{array}{l}\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B \\ \qquad \text{then} \ \langle \Box, A, S, B, T\rangle_n \\ \qquad \text{else} \ \ \text{if } \textit{propagate-condition} \\ \qquad\qquad\qquad\qquad \text{then} \ \ \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S', B', T'\rangle_{n'}) \\ \qquad\qquad\qquad\qquad \text{else} \ \ \langle \Box, A, S, B, T\rangle_n\end{array}$$

where *propagate-condition* is that there exists a matching substitution θ such that

$$\left\{\begin{array}{l} S = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus R \\ c = \theta(d) \\ chr(H_1) = \theta(H_1') \wedge chr(H_2) = \theta(H_2') \wedge chr(H_3) = \theta(H_3') \\ \mathcal{D}_b \models B \rightarrow \bar{\exists}_{\theta(r)}\theta(g) \\ t = id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T \\ \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), [c\#i : j|A], S \setminus H_3, B, T \cup \{t\}\rangle_n) = \langle \Box, [c\#i : j|A], S', B', T'\rangle_{n'} \end{array}\right.$$

where g and $\theta(g)$, respectively C and $\theta(C)$ are variants and $vars(g) \cap vars(\theta(g)) = \emptyset$ and $vars(C) \cap vars(\theta(C)) = \emptyset$.

6. Goal

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle\Box, A, S, B, T\rangle_n) = \\ \quad \langle\Box, A, S, B, T\rangle_n \\ \mathcal{S}[\![\mathcal{P}]\!](\langle[c|C], A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B' \\ \qquad \text{then } \langle\Box, A, S', B', T'\rangle_{n'} \\ \qquad \text{else } \mathcal{S}[\![\mathcal{P}]\!](\langle C, A, S', B', T'\rangle_{n'})\end{array}$$

where $[c|C]$ is a sequence of built-in and CHR constraints and

$$\mathcal{S}[\![\mathcal{P}]\!](\langle c, A, S, B, T\rangle_n) = \langle\Box, A, S', B', T'\rangle_{n'}$$

For the first propagation step above, the result of the auxiliary function $\mathcal{S}_{\text{Prop}}$ is used:

$$\begin{array}{lll}\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1 : 1, [], \{p\#1\}, \mathit{true}, \bullet\rangle_2) & & \\ \quad = \quad \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1 : 1, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) & & \\ \quad = \quad \langle\Box, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3 & & \end{array}$$

The second step in the evaluation of $\mathcal{S}_{\text{Prop}}$ is obtained through the evaluation of:

$$\begin{array}{lll}\mathcal{S}[\![\mathcal{P}]\!](\langle q, [p\#1 : 1], \emptyset, \{p\#1\}, \emptyset\rangle_2) & & (\textbf{Activate}) \\ \quad = \quad \mathcal{S}[\![\mathcal{P}]\!](\langle q\#2 : 1, [p\#1 : 1], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) & & (\textbf{Drop}) \\ \quad = \quad \langle\Box, [p\#1 : 1], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3 & & \end{array}$$

The last propagation step in the main computation is obtained through a similar evaluation of $\mathcal{S}_{\text{Prop}}$:

$$\begin{array}{lll}\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1 : 4, [], \{p\#1, q\#2\}, \mathit{true}, \bullet\rangle_3) & & \\ \quad = \quad \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle p\#1 : 4, [], \{q\#2\}, \mathit{true}, \bullet\rangle_4) & & \\ \quad = \quad \langle\Box, [p\#1 : 4], \{q\#2\}, \mathit{true}, \bullet\rangle_4 & & \end{array}$$

The second step in the above evaluation of $\mathcal{S}_{\text{Prop}}$ is obtained through:

$$\begin{array}{lll}\mathcal{S}[\![\mathcal{P}]\!](\langle s, [p\#1 : 4], \emptyset, \{p\#1, q\#2\}, \emptyset\rangle_3) & & (\textbf{Activate}) \\ \quad = \quad \mathcal{S}[\![\mathcal{P}]\!](\langle s\#3 : 1, [p\#1 : 4], \{p\#1, q\#2, s\#3\}, \mathit{true}, \bullet\rangle_4) & & (\textbf{Simplify}) \\ \quad = \quad \mathcal{S}[\![\mathcal{P}]\!](\langle\Box, [p\#1 : 4], \{q\#2\}, \mathit{true}, \bullet\rangle_4) & & (\textbf{Goal}) \\ \quad = \quad \langle\Box, [p\#1 : 4], \{q\#2\}, \mathit{true}, \bullet\rangle_4 & & \end{array}$$

9.3 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 9.3.1 and 9.3.2 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 9.3.3. It covers how the framework applies the abstract semantic function starting from an initial state and how the framework deals with non-determinism.

9.3.1 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 9.2.1, 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.

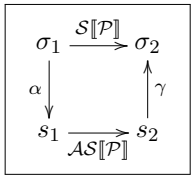
The framework will only make use of the least upper bound operation $s_1 \sqcup s_2$ on states corresponding to the same program point ($\mathsf{pp}(s_1) = \mathsf{pp}(s_2)$). Similarly, α is only applied to a set of concrete states corresponding to the same program point. Moreover, we will only explicitly define α for a single concrete state σ. The extension of α to a set S of concrete states is assumed to be:

$$\alpha(S) = \bigsqcup_{\sigma \in S} \alpha(\sigma)$$

9.3.2 Abstract Semantic Function

The abstract domain must provide an abstract semantic function $\mathcal{AS} : \mathbf{Prog} \to (\mathbf{\Sigma_a} \hookrightarrow \mathbf{\Sigma_a})$ 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 9.2.2.

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:

$$\forall S \subseteq \mathbf{\Sigma} : \{\mathcal{S}[\![\mathcal{P}]\!](\sigma) | \sigma \in S\} \subseteq \gamma \circ \mathcal{AS}[\![\mathcal{P}]\!] \circ \alpha(S)$$

9.3.3 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 $\mathsf{new\_goal}$ 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 9.5.

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.

9.4 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 9.4.1 we define the property that the analysis derives. Next, the abstract domain and abstract semantic function of the analysis are defined in Sections 9.4.2 and 9.4.3 respectively. Section 9.4.4 illustrates the application of the analysis on a small program.

9.4.1 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 9.1 (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

$$\begin{array}{l}\mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \textit{simplify-condition} \\ \quad\quad \text{then} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), obs(A, H_1 \cup H_2 \cup H_3), H_1 \uplus S', \theta \wedge B, T \cup \{t\}\rangle_n) \\ \quad\quad \text{else} \quad \mathcal{S}[\![\mathcal{P}]\!](\langle c\#i : j+1, A, S, B, T\rangle_n)\end{array}$$

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:

$$\begin{array}{l}\mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A, S, B, T\rangle_n) = \\ \quad \text{if } \mathcal{D}_b \models \neg\bar{\exists}_{\emptyset} B \\ \quad\quad \text{then} \ \langle \Box, A, S, B, T\rangle_n \\ \quad\quad \text{else} \ \ \text{if } \textit{propagate-condition} \\ \quad\quad\quad\quad\quad\quad \text{then} \ \mathcal{S}_{\text{Prop}}[\![\mathcal{P}]\!](\langle c\#i : j, A', S', B', T'\rangle_{n'}) \\ \quad\quad\quad\quad\quad\quad \text{else} \ \ \langle \Box, A, S, B, T\rangle_n\end{array}$$

where *propagate-condition* is that there exists a matching substitution θ such that

$$\left\{\begin{array}{l} S = \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus R \\ c = \theta(d) \\ chr(H_1) = \theta(H_1') \wedge chr(H_2) = \theta(H_2') \wedge chr(H_3) = \theta(H_3') \\ \mathcal{D}_b \models B \rightarrow \bar{\exists}_{\theta(r)}\theta(g) \\ t = id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T \\ \mathcal{S}[\![\mathcal{P}]\!](\langle \theta(C), [c\#i : j | obs(A, H_1 \cup H_2 \cup H_3)], S \setminus H_3, B, T \cup \{t\}\rangle_n) = \\ \qquad \langle \Box, [E|A'], S', B', T'\rangle_{n'} \\ E \in \{c\#i : j, c\#i : j^*\} \end{array}\right.$$

where g and $\theta(g)$, respectively C and $\theta(C)$ are variants and $vars(g) \cap vars(\theta(g)) = \emptyset$ and $vars(C) \cap vars(\theta(C)) = \emptyset$.

Example 9.2 When examining the evaluation shown in Section 9.2.3 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

$$\mathcal{S}[\![\mathcal{P}]\!](\langle s\#3:1,[p\#1:4],\{p\#1,q\#2,s\#3\},\mathit{true},\bullet\rangle_4) \qquad \textbf{(Simplify)}$$
$$= \quad \mathcal{S}[\![\mathcal{P}]\!](\langle\Box,[p\#1:4^*],\{q\#2\},\mathit{true},\bullet\rangle_4)$$

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.

9.4.2 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:

$$\begin{aligned}
\alpha_{ls}(c) &= \texttt{builtin} && (c \text{ built-in})\\
\alpha_{ls}(p(t_1,\ldots,t_n)) &= p\\
\alpha_{ls}(p(t_1,\ldots,t_n)\#i) &= p\\
\alpha_{ls}(p(t_1,\ldots,t_n)\#i:j) &= p:j\\
\alpha_{ls}([]) &= []\\
\alpha_{ls}([c|G]) &= [\alpha_{ls}(c)|\alpha_{ls}(G)]\\
\alpha_{ls}(S) &= \{\alpha_{ls}(c)|c\in S\} && (S \text{ set})\\
\alpha_{ls}(\langle G,A,_,_,_\rangle__) &= \langle\alpha_{ls}(G),\alpha_{ls}(\mathsf{unobserved}(A))\rangle
\end{aligned}$$

where $\mathsf{unobserved}$ is defined as

$$\mathsf{unobserved}(A)=\left\{p \;\middle|\; \begin{array}{l} p(t_1,\ldots,t_n)\#i:j\in\mathsf{list2set}(A),\\ \neg\exists p(t'_1,\ldots,t'_n)\#i':j'^*\in\mathsf{list2set}(A)\end{array}\right\}$$

$$\begin{aligned}
\mathsf{list2set}([]) &= []\\
\mathsf{list2set}([a|A]) &= \{a\}\cup\mathsf{list2set}(A)
\end{aligned}$$

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 $\mathsf{unobserved}$.

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:

$$
\begin{array}{l}
s_1 \sqcup_{ls} s_2 = \\
\quad \text{if } s_1 = \langle G, A_1 \rangle \wedge s_2 = \langle G, A_2 \rangle \\
\quad\quad \text{then } \langle G, (A_1 \cap A_2) \rangle \\
\quad\quad \text{else } \top_{ls}
\end{array}
$$

9.4.3 Abstract Semantic Function

The abstract semantic function $\mathcal{AS}$ for the late storage domain is defined below.

AbstractSolve

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \texttt{builtin}, A \rangle_n) = \langle \Box, \emptyset \rangle$$

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: $\mathcal{AS}[\![\mathcal{P}]\!](\langle c, \emptyset \rangle) = \langle \Box, \emptyset \rangle$.

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

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c : j, A \rangle) = \langle \Box, A \rangle_n$$

Applicable if no occurrence j exists for CHR constraint c in $\mathcal{P}$.

AbstractGoal

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle [c_{k_1}, \ldots, c_{k_n}], A \rangle) = \langle \Box, A' \rangle_n$$

where

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c_{k_i}, A \rangle) = \langle \Box, A_i \rangle$$

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.

9.4.4 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.

9.5 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 9.5.1 and 9.5.2 present the abstract domain and the abstract semantic function respectively. The analysis is illustrated by means of an example in Section 9.5.3.

9.5.1 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.

$$
\begin{array}{rcl}
\alpha_g(c) & = & c \ (c \text{ is built-in}) \\
\alpha_g(p(t_1,\ldots,t_n)) & = & p(t_1,\ldots,t_n) \\
\alpha_g(p(t_1,\ldots,t_n)\#i) & = & p(t_1,\ldots,t_n) \\
\alpha_g(p(t_1,\ldots,t_n)\#i:j) & = & p(t_1,\ldots,t_n):j
\end{array}
$$

$$
\begin{array}{rcll}
\alpha_g([]) & = & [] & \\
\alpha_g([c|G]) & = & [\alpha_g(c)|\alpha_g(G)] & \\
\alpha_g(S) & = & \{\alpha_g(c)|c \in S\} & (S \text{ set or multiset})
\end{array}
$$

$$
\begin{array}{l}
\alpha_g(p(t_1,\ldots,t_n)\#i, B) = p(x_1,\ldots,x_n) \leftarrow D \\
\quad \text{where } D = \bar{\exists}^{\mathcal{G}}_{x_1,\ldots,x_n} \alpha_{\mathcal{G}}(B \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n)
\end{array}
$$

$$
\begin{array}{l}
\alpha_g(S, B) = \mathit{snf}(\{\alpha_g(c, B)|c \in S\}) \ \ (S \text{ set or multiset}) \\
\alpha_g(\langle G, \_, S, B, \_\rangle_\_) = \langle \alpha_g(G), \alpha_g(S, B), \bar{\exists}^{\mathcal{G}}_{vars(G)} \alpha_{\mathcal{G}}(B)\rangle
\end{array}
$$

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:

$$
\begin{array}{rcl}
\mathit{snf}(\emptyset) & = & \emptyset \\
\mathit{snf}(\{p(\bar{x}) \leftarrow D_1\} \uplus S) & = & \mathit{snf}(\{p(\bar{x}) \leftarrow D_1 \sqcup_{\mathcal{G}} D_2\} \uplus S') \\
 & & \text{where } S = \{p(\bar{x}) \leftarrow D_2\} \uplus S' \\
\mathit{snf}(\{p(\bar{x}) \leftarrow D_1\} \uplus S) & = & \{p(\bar{x}) \leftarrow D_1\} \uplus \mathit{snf}(S), \\
 & & \text{where } \neg\exists p(\bar{x}) \leftarrow D_2 \in S
\end{array}
$$

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 $\preceq_g$. It follows form the definition of partial ordering that all variants of the same abstract execution state are considered equal.

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.

9.5.2 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

$$\begin{array}{lcll} S_0 & = & S_a \uplus S_b & \\ s_j & = & \langle c, S_a \uplus S_b, B\rangle & , j = 0 \\ s_j & = & \langle \Box, S_j, \_\rangle = \bigsqcup_g \{s_j^i \mid s_j^i = \mathcal{AS}[\![\mathcal{P}]\!](\langle p_i(\bar{x}_i), S_{j-1}, D_i\rangle) \wedge 1 \leq i \leq n\}, & j \geq 1 \end{array}$$

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.

9.5.3 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, 2005b], 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:

$$\begin{aligned} S_1 &= \{main\texttt{:-}\emptyset\} \\ S_2 &= S_1 \cup \{candidate(N)\texttt{:-}\{N\}\} \\ S_3 &= S_2 \cup \{prime(N)\texttt{:-}\{N\}\} \end{aligned}$$

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle main, \emptyset, \emptyset\rangle) && \text{(AbstractActivate)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle main : 1, S_1, \emptyset\rangle) && \text{(AbstractSimplify)} \\ &\quad = \langle \Box, S_3, \emptyset\rangle \sqcup_g \langle \Box, S_1, \emptyset\rangle \\ &\quad = \langle \Box, S_3, \emptyset\rangle \end{aligned}$$

In order to obtain the above abstract simplification result, the following evaluation of the first rule's body is needed:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle [X = 10, candidate(X)], S_1, \emptyset\rangle) && \text{(AbstractGoal)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [candidate(X)], S_1, \{X\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \langle \Box, S_3, \{X\}\rangle \end{aligned}$$

For the first abstract goal step, this auxiliary result is used:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle X = 10, \emptyset, S_1\rangle) && \text{(AbstractSolve)} \\ &\quad = \langle \Box, S_1, \{X\}\rangle \end{aligned}$$

For the second abstract goal step, this auxiliary result is used:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X), S_1, \{X\}\rangle) && \text{(AbstractActivate)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X) : 1, S_2, \{X\}\rangle) && \text{(AbstractSimplify)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \Box, S_2, \{X\}\rangle) \sqcup_g \mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X) : 2, S_2, \{X\}\rangle) \\ &\quad = \langle \Box, S_2, \{X\}\rangle \sqcup_g \langle \Box, S_3, \{X\}\rangle \\ &\quad = \langle \Box, S_3, \{X\}\rangle \end{aligned}$$

which uses the result:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle candidate(X) : 2, S_2, \{X\}\rangle) && \text{(AbstractSimplify)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [prime(X), Y \ is \ X - 1, candidate(Y)], S_2, \{X\}\rangle) \sqcup_g \langle \Box, S_2, \{X\}\rangle) \\ &\quad = \langle \Box, S_3, \{X\}\rangle \sqcup_g \langle \Box, S_2, \{X\}\rangle \\ &\quad = \langle \Box, S_3, \{X\}\rangle \end{aligned}$$

This in turn uses:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle [prime(X), Y \ is \ X - 1, candidate(Y)], S_2, \{X\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [Y \ is \ X - 1, candidate(Y)], S_3, \{X\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle [candidate(Y)], S_3, \{X, Y\}\rangle) && \text{(AbstractGoal)} \\ &\quad = \langle \Box, S_3, \{X, Y\}\rangle \end{aligned}$$

The evaluation for the `prime(X)` goal is as follows:

$$\begin{aligned} &\mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N), S_2, \{N\}\rangle) && \text{(AbstractActivate)} \\ &\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N) : 1, S_3, \{N\}\rangle) && \text{(AbstractSimplify)} \\ &\quad = \langle \Box, S_3, \{N\}\rangle \sqcup_g \mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N) : 2, S_3, \{N\}\rangle) && \text{(AbstractPropagate)} \\ &\quad = \langle \Box, S_3, \{N\}\rangle \sqcup_g \mathcal{AS}[\![\mathcal{P}]\!](\langle prime(N) : 3, S_3, \{N\}\rangle) && \text{(AbstractDrop)} \\ &\quad = \langle \Box, S_3, \{N\}\rangle \sqcup_g \langle \Box, S_3, \{N\}\rangle) \\ &\quad = \langle \Box, S_3, \{N\}\rangle \end{aligned}$$

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.

9.6 Implementation and Evaluation

We have implemented both the late storage analysis and the groundness analysis in our K.U.Leuven CHR system (see [Schrijvers, 2005a, 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.

9.6.1 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 [Schrijvers, 2005a, Chapter 6] for a more extensive treatment of late storage.

  With our late storage analysis, the optimization of [Schrijvers, 2005a, Chapter 6] 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 occurrence 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 9.2 we show the speed-ups resulting from late storage analysis in hProlog. For eight benchmarks, see [Schrijvers, 2005b], 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 9.2: Late storage analysis: runtime results of optimized programs relative to unoptimized programs

In Table 9.3 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 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 9.3: 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 to 20 ms for the above benchmarks and mostly only a fraction of total compilation time. For

[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.

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.

9.6.2 Groundness Analysis

Our implementation of groundness analysis uses the naive groundness domain for built-in constraints as it is described in Section 9.5.3.

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 [Schrijvers, 2005a, 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 3.5.3: `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 constraints in those benchmarks. The speed-ups realized by the annotations were listed in Table 3.2 in Section 3.5.3.

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.

9.7 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 et al., 2005, Schrijvers and Demoen, 2004a] (see [Schrijvers, 2005a, Section 6.3.4]), late storage, continuation optimization, an index optimization [Holzbaur et al., 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 et al., 2004a] and accepted at the Principles and Practice of Declarative Programming Symposium [Schrijvers et al., 2005].

9.7.1 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 et al., 1990, Codish et al., 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.

Bibliography

Michael Codish, Moreno Falaschi, Kim Marriott, and William H. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. In *ICALP'93: Proceedings of the 20th International Colloquium on Automata, Languages and Programming*, pages 633–644, London, UK, 1993. Springer Verlag.

C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In S. Debray and M. Hermenegildo, editors, *NACLP'90: Proceedings of the*

North American Conference on Logic Programming, pages 215–232, Cambridge, MA, USA, 1990. MIT Press.

Patrick Cousot and Radhia Cousot. 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*, pages 238–252, Los Angeles, California, 1977. ACM Press.

Gregory Duck, Tom Schrijvers, and Peter Stuckey. Abstract Interpretation for Constraint Handling Rules. Report CW 391, K.U.Leuven, Department of Computer Science, Leuven, Belgium, September 2004a.

Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, St-Malo, France, September 2004b. Springer Verlag.

Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5): 503–531, 2005.

Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. PO Box 1263, SE-164 29 Kista, Sweden, October 2003.

Kim Marriott, Harald Søndergaard, and Neil D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994. ISSN 0164-0925.

Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, Department of Computer Science, K.U.Leuven, Belgium, 2005a.

Tom Schrijvers. A Collection of Assorted CHR Benchmarks, 2005b. http://www.cs.kuleuven.be/~toms/Research/CHR/.

Tom Schrijvers and Bart Demoen. Antimonotony-based delay avoidance for CHR. Report CW 385, K.U.Leuven, Department of Computer Science, July 2004a.

Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, Ulm, Germany, May 2004b.

Tom Schrijvers, Peter Stuckey, and Gregory Duck. Abstract Interpretation for Constraint Handling Rules. In *PPDP'05: Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming*, pages 218–229, Lisbon, Portugal, July 2005. ACM Press.

Index