

The Computational Power and Complexity of Constraint Handling Rules

JON SNEYERS, TOM SCHRIJVERS and BART DEMOEN
K.U.Leuven, Belgium

Constraint Handling Rules (CHR) is a high-level rule-based programming language which is increasingly used for general purposes. We introduce the CHR machine, a model of computation based on the operational semantics of CHR. Its computational power and time complexity properties are compared to those of the well-understood Turing machine and Random Access Memory machine. This allows us to prove the interesting result that every algorithm can be implemented in CHR with the best known time and space complexity. We also investigate the practical relevance of this result and the constant factors involved. Finally we expand the scope of the discussion to other (declarative) programming languages.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*

General Terms: Languages, Algorithms, Theory, Performance, Experimentation

Additional Key Words and Phrases: Constraint Handling Rules, complexity, constant factors

1. INTRODUCTION

Constraint Handling Rules (CHR) is a high-level programming language based on multi-headed committed-choice rules, introduced by Frühwirth [1992; 1998; 2008]. Originally designed for writing constraint solvers, CHR is now used as a general-purpose programming language. When new programming languages are introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. Schrijvers and Frühwirth [2006] and Sneyers et al. [2006a] have discussed case studies: they describe efficient and elegant CHR implementations of, respectively, Tarjan's union-find algorithm and Dijkstra's shortest path algorithm, indicating an affirmative answer. In contrast, it is not clear whether these algorithms can be implemented with optimal complexity in pure Prolog.

This paper tackles the above question in a more general way. The major contribution of this paper is the proof of a general affirmative answer: we demonstrate

A preliminary version of this article was presented at the *2nd Workshop on Constraint Handling Rules (CHR'05)* which was held on October 5, 2005 in Sitges, Spain at the occasion of ICLP'05. Authors' address: Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium; email: {jon,toms,bmd}@cs.kuleuven.be.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2008 ACM 0164-0925/2008/0500-0001 \$5.00

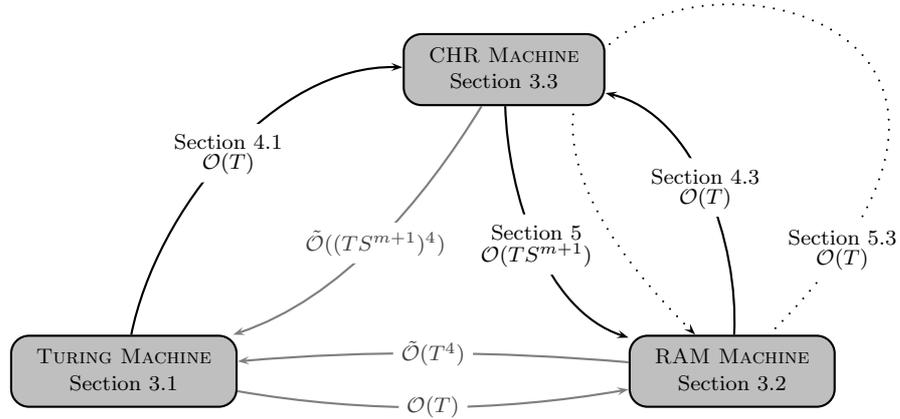


Fig. 1. Overview. An arrow between A and B indicates that A can be simulated on B . Labels indicate the relevant section and the time complexity of simulating a T -time, S -space A on B .

that it is possible to implement *any* algorithm in CHR in an efficient way, i.e. with the best known time and space complexity. Our approach is as follows. We introduce a new model of computation, the CHR machine, and compare it with the well-known Turing machine and RAM machine models. We show how to program the CHR machine to efficiently simulate both other machines — in particular the RAM machine, which is faster than the Turing machine. We then investigate how to simulate a CHR machine efficiently on a RAM machine using an optimizing CHR compiler. This results in a general complexity meta-theorem. Finally we apply this meta-theorem to the RAM machine simulator. This allows us to conclude that existing CHR compilation techniques suffice to implement the RAM machine simulator efficiently. As a result, *all* RAM machine programs (so also every known algorithm) can be translated to CHR programs with the same complexity. Figure 1 gives an overview of the relation between the three models of computation. Note that the $\tilde{\mathcal{O}}((TS^{m+1})^4)$ 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.

Notational conventions

Prolog list notation is used to denote sequences: $[h|T] = [h] ++ T$ where $++$ denotes sequence concatenation. Σ^* denotes the set of all sequences of elements of Σ . The notation $vars(t)$ denotes the set of variables used in term t . The abbreviation $\exists_{\{v_1, \dots, v_n\}}$ means $\exists v_1 \dots \exists v_n$. The notation $\exists_V F$ is used to denote $\exists_{vars(F) \setminus V} F$, that is quantifying over all variables not in V . Finally, $\tilde{\mathcal{O}}(T)$ denotes $\mathcal{O}(TL)$, where L is polylogarithmic in the variables of T . For example, a complexity of $\mathcal{O}(n^2 \log^2 n \log \log n)$ could be denoted by $\tilde{\mathcal{O}}(n^2)$. Note that a function which is $\tilde{\mathcal{O}}(n^k)$ is also $\mathcal{O}(n^{k+\epsilon})$ for arbitrarily small $\epsilon > 0$.

Overview

In the next section we briefly recapture syntax and semantics of CHR. Section 3 defines the three models of computation: Turing machines, RAM machines and

CHR machines. Next, in Section 4, we investigate the theoretical relation between CHR machines and the other computational models. In Section 5 we discuss compilation of CHR programs, linking the theoretical CHR machines and practical CHR systems. This concludes the proof of our main result. Section 6 investigates, experimentally, the practical relevance of the result and the constant factors of the complexities. In Section 7 we try to ‘port’ the results to other declarative languages. Section 8 concludes this paper.

2. CONSTRAINT HANDLING RULES

In this section we introduce the syntax (subsection 2.1) and semantics (subsection 2.2) of Constraint Handling Rules. Gentler introductions to CHR can be found in [Frühwirth 1998], [Schrijvers 2005], [Frühwirth 2008], and [Sneyers et al. 2008]. Readers that are already familiar with CHR and its theoretical operational semantics may skip to subsection 2.3, in which some new terminology is introduced.

2.1 Syntax

We assume the reader to be familiar with some basic notions of Constraint Logic Programming [Marriott and Stuckey 1998; Frühwirth and Abdennadher 2003].

CHR is embedded in a host language \mathcal{H} that provides data types and a number of predefined constraints. These constraints are called host language constraints or *built-in* constraints. The typical host language of CHR is Prolog. Its host language constraints are unification, Prolog built-ins and other (user-defined) Prolog predicates. Its data types are Prolog variables and terms.

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

We assume the host language to offer at least one data type that can be used as an identifier; for instance Prolog variables (without unification). We assume the built-in constraint theory (denoted by $\mathcal{D}_{\mathcal{H}}$) to define at least the basic constraints *true* and *fail* and the *ask*-versions of syntactic equality (“==”) and inequality (“\==”).

We denote the set of multisets of (CHR and host language) constraints 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 an ordered sequence of CHR rules. There are three different kinds of rules. A *simplification* rule is of the form:

$$c_1, \dots, c_n \iff g \mid d_1, \dots, d_m.$$

A *propagation rule* is of the form:

$$c_1, \dots, c_n \implies g \mid d_1, \dots, d_m.$$

A *simplification rule* is of the form:

$$c_1, \dots, c_l \setminus c_{l+1}, \dots, c_n \iff g \mid d_1, \dots, d_m.$$

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

```

loop @ upto(N) ⇔ N > 1 | prime(N), N1 is N - 1, upto(N1).
stop @ upto(1) ⇔ true.
absorb @ prime(A) \ prime(B) ⇔ B mod A =: 0 | true.

```

Fig. 2. The CHR(Prolog) program PRIMES, a prime number sieve.

rule. A rule is optionally preceded by *name* @ where *name* is a term. No two rules may have the same name. Rules without a name get an unique implicit name.

All the head constraints of a simplification rule and the head constraints c_{l+1}, \dots, c_n of a simpagation rule are called *removed* head constraints. The other head constraints — all heads of a propagation rule and c_1, \dots, c_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 d_1, \dots, d_n 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, g , is called the *guard*. It is a conjunction of host language ask constraints. The guard “ g | ” is optional; if omitted, it is considered to be “*true* | ”.

For simplicity, we sometimes consider both simplification and propagation rules as special cases of a simpagation rule. We often use the following notation to denote any kind of rule:

$$H_k \setminus H_r \iff g \mid B$$

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.

Example. Figure 2 lists a simple CHR(Prolog) program called PRIMES, a CHR variant of the Sieve of Eratosthenes. Dating back to at least 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. Note that 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 **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 .

2.2 The ω_t operational semantics

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

The 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.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$
 where c is a built-in constraint and $\mathcal{D}_{\mathcal{H}} \models \exists_0 \mathbb{B}$.
2. **Introduce.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
 where c is a CHR constraint and $\mathcal{D}_{\mathcal{H}} \models \exists_0 \mathbb{B}$.
3. **Apply.** $\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle C \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
 where \mathcal{P} contains a (renamed apart) rule of the form $r @ H'_1 \setminus H'_2 \iff g \mid C$
 and θ is a matching substitution such that $\text{chr}(H_1) = \theta(H'_1)$ and $\text{chr}(H_2) = \theta(H'_2)$
 and $h = (r, \text{id}(H_1), \text{id}(H_2)) \notin T$ and $\mathcal{D}_{\mathcal{H}} \models (\exists_0 \mathbb{B}) \wedge (\mathbb{B} \rightarrow \exists_{\mathbb{B}}(\theta \wedge g))$.

Fig. 3. The transition rules of the theoretical operational semantics ω_t , defining $\mapsto_{\mathcal{P}}$.

Definition 2.1. 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} . An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i , the *constraint identifier*. This number serves to differentiate among copies of the same constraint. We introduce the functions $\text{chr}(c\#i) = c$ and $\text{id}(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner, e.g. $\text{id}(\mathbb{S}) = \{c\#i \mid c\#i \in \mathbb{S}\}$. Note that $\text{chr}(\mathbb{S})$ is a multiset although \mathbb{S} is a set. 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. This is necessary to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used to fire the same rule before. Finally, the counter $n \in \mathbb{N}$ represents the next free integer that can be used to number a CHR constraint. We use $\sigma, \sigma_0, \sigma_1, \dots$ to denote execution states and Σ^{CHR} to denote the set of all execution states.

Transitions are defined by the binary relation $\mapsto_{\mathcal{P}}: \Sigma^{\text{CHR}} \rightarrow \Sigma^{\text{CHR}}$ shown in Figure 3. Execution proceeds by exhaustively applying the transition rules, starting from an initial state. We define $\mapsto_{\mathcal{P}}^*$ as the transitive closure of $\mapsto_{\mathcal{P}}$.

2.3 Derivations

Definition 2.2. Given an *initial goal* (or *query*) $\mathbb{G} \in \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$, the *initial state* is $\text{initstate}(\mathbb{G}) = \langle \mathbb{G}, \emptyset, \text{true}, \emptyset \rangle_1$. The set of initial states is denoted by $\Sigma^{\text{init}} \subset \Sigma^{\text{CHR}}$.

Definition 2.3. 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: $\neg \exists \sigma \in \Sigma^{\text{CHR}} : \sigma_f \mapsto_{\mathcal{P}} \sigma$. In a *failure* state, the underlying solver \mathcal{H} can prove $\mathcal{D}_{\mathcal{H}} \models \neg \exists_0 \mathbb{B}$ — such states are always final. A *successful* final state is a final state that is not a failure state, i.e. $\mathcal{D}_{\mathcal{H}} \models \exists_0 \mathbb{B}$. The set of final states is denoted by $\Sigma^{\text{final}} \subset \Sigma^{\text{CHR}}$.

Definition 2.4. Given a CHR program \mathcal{P} , a *finite derivation* d is a finite sequence $[\sigma_0, \sigma_1, \dots, \sigma_n]$ of states where $\sigma_0 \in \Sigma^{\text{init}}$, $\sigma_n \in \Sigma^{\text{final}}$, and $\sigma_i \mapsto_{\mathcal{P}} \sigma_{i+1}$ for $0 \leq i < n$. If σ_n is a failure state, we say d has *failed*, otherwise d is a *successful* derivation.

Definition 2.5. An *infinite derivation* d_{∞} is an infinite sequence $\sigma_0, \sigma_1, \dots$ of states where $\sigma_0 \in \Sigma^{\text{init}}$ and $\sigma_i \mapsto_{\mathcal{P}} \sigma_{i+1}$ for $i \in \mathbb{N}$.

<i>reflexivity</i>	@ $\mathbf{leq}(X,X)$	$\iff true.$
<i>antisymmetry</i>	@ $\mathbf{leq}(X,Y), \mathbf{leq}(Y,X)$	$\iff X = Y.$
<i>idempotence</i>	@ $\mathbf{leq}(X,Y) \setminus \mathbf{leq}(X,Y)$	$\iff true.$
<i>transitivity</i>	@ $\mathbf{leq}(X,Y), \mathbf{leq}(Y,Z)$	$\implies \mathbf{leq}(X,Z).$

Fig. 4. The CHR(Prolog) program LEQ, a solver for the less-than-or-equal relation.

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 $\Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$ 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 $\rightsquigarrow_{\Delta}: \Sigma^{init} \rightarrow \Sigma^{CHR} \cup \{\infty\}$:

Definition 2.6. State σ_n is a Δ -output of σ_0 if $[\sigma_0, \dots, \sigma_n] \in \Delta$. 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 2.7. The CHR program \mathcal{P} is Δ -deterministic for input $I \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ if the restriction of $\rightsquigarrow_{\Delta}$ to $initstate[I]$ is a function and $\forall i \in I, d \in \Delta|_i : \text{if } d \text{ is a successful derivation, then } \forall d' \in \Delta|_i : \#d = \#d'$.

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. Note that if a program \mathcal{P} is $\Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$ -deterministic for all input $\mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$, it is also observable confluent [Duck et al. 2006]. The notion of observable confluence does not require derivations to have the same length.

Example. Consider the CHR program of the previous example. This program (PRIMES, listed in Figure 2) is $\Delta_{\omega_t}^{\text{Prolog}}$ -deterministic for input $\{\mathbf{upto}(n) | n \in \mathbb{N}\}$. 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 $\Delta_{\omega_t}^{\mathcal{H}}$ -deterministic for its intended input:

Example. The CHR program LEQ listed in Figure 4 is not $\Delta_{\omega_t}^{\text{Prolog}}$ -deterministic for all conjunctions of $\mathbf{leq}/2$ constraints. Consider the input query “ $\mathbf{leq}(A,B), \mathbf{leq}(B,A)$ ”. One derivation consists of two **Introduce** steps followed by an **Apply** step using the *antisymmetry* rule, followed by a **Solve** step for “ $A = B$ ” and two **Apply** steps using the *reflexivity* rule. This derivation has length six. In another derivation, the *transitivity* rule is applied after the **Introduce** steps. This results in a longer derivation, or even an infinite derivation.

For the connoisseurs: LEQ is not even $\Delta_{\omega_r}^{\text{Prolog}}$ -deterministic (i.e. restricting the derivations to those of the refined operational semantics ω_r , cfr. Section 5.2.1). Given the query “ $\mathbf{leq}(B,C), \mathbf{leq}(B,A), \mathbf{leq}(A,B)$ ”, the derivation length depends on which partner for $\mathbf{leq}(A,B)$ is tried first when the *transitivity* rule is applied.

3. MODELS OF COMPUTATION

Both computability theory and computational complexity theory define idealized *models of computation*. In this section we define three different models: the well-known Turing machine (subsection 3.1); the RAM machine (subsection 3.2), which more closely models realistic computers; and finally the *CHR machine* (subsection 3.3), which we will use to study the CHR programming language.

3.1 Turing machines

The Turing machine, originally introduced by Alan Turing [1936], is the prototypical computational model used in computability and complexity theory. We will use a single-tape definition which corresponds to the one given by Hopcroft et al. [2001].

Definition 3.1. A *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
- $\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{\mathbf{left}, \mathbf{right}\}$ is a partial function

The function δ is called the *transition function*, “**left**” represents *left shift* and “**right**” represents *right shift*.

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 $X = \mathbf{left}$ or $X = \mathbf{right}$.
- 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*.

Formally, we represent an execution state of a Turing machine $M = \langle Q, \Sigma, q_0, b, F, \delta \rangle$ as a 4-tuple $\sigma^{\text{TM}} = \langle q, P, s, N \rangle$, 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 $\sigma_0^{\text{TM}}, \sigma_1^{\text{TM}}, \dots$ to denote Turing machine execution states, and $\Sigma^{\text{TM}}(M)$ 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 $\sigma_i^{\text{TM}} = \langle q_0, [], f, N \rangle$. The

transition rules are:

$$\begin{array}{llll}
 (1) & \langle q, [p|P], s, N \rangle \rightarrow \langle q', P, p, [s'|N] \rangle & \text{if } \delta(q, s) = (q', s', \text{left}) \\
 (2) & \langle q, P, s, [n|N] \rangle \rightarrow \langle q', [s'|P], n, N \rangle & \text{if } \delta(q, s) = (q', s', \text{right}) \\
 (3) & \langle q, [], s, N \rangle \rightarrow \langle q', [], b, [s'|N] \rangle & \text{if } \delta(q, s) = (q', s', \text{left}) \\
 (4) & \langle q, P, s, [] \rangle \rightarrow \langle q', [s'|P], b, [] \rangle & \text{if } \delta(q, s) = (q', s', \text{right})
 \end{array}$$

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 $\sigma_i^{\text{TM}} = \langle q, P, s, N \rangle$, the contents of the tape is the sequence $\text{tape}(\sigma_i^{\text{TM}}) = \text{reverse}(P) ++ [s] ++ N$, where reverse reverses a sequence (i.e. $\text{reverse}([a_1, a_2, \dots, a_{n-1}, a_n]) = [a_n, a_{n-1}, \dots, a_2, a_1]$). 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*. The *output* of the Turing machine is the tape contents in the final execution state if the input is accepted, and undefined if the input is rejected. Given a Turing machine M and input I , we denote the corresponding derivation by $\text{deriv}_M(I)$. If the machine terminates on input I , we denote the output by $M(I) = \text{tape}(\sigma_f^{\text{TM}})$, where σ_f^{TM} is the last state in $\text{deriv}_M(I)$.

3.2 Random Access Memory machines

The Random Access Memory (RAM) machine closely models the basic features of traditional sequential computers. In the literature many variations of the RAM 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, \dots to represent the memory addresses and $\llbracket A \rrbracket$ to denote the value of the register at address A . 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 is written such that all additional registers are initialized (using the **init** instruction) before they are used.¹

The program consists of a sequence of instructions. The program instructions are labeled with successive natural numbers. We use L, L_1, \dots 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

¹A program that does not initialize registers before it uses them can be rewritten in the following way. We may assume all registers that are used directly in operands to be initialized in advance (these can be easily found by inspecting the program). We let the program instructions that use indirect addressing (**imv** and **mvi**) be preceded by an **init** instruction. For example, the instruction "**mvi** a b " should be rewritten to "**init** b ; **mvi** a b ". This transformation at most doubles the program size and the number of instructions executed.

Table I. Instruction set of the RAM

<i>Instruction</i>			<i>Effect</i>
inc	A		$\llbracket A \rrbracket \leftarrow \llbracket A \rrbracket + 1$
dec	A		$\llbracket A \rrbracket \leftarrow \llbracket A \rrbracket - 1$
clr	A		$\llbracket A \rrbracket \leftarrow 0$
jmp	L		$PC \leftarrow L$
cjmp	A	L	$PC \leftarrow L$ if $\llbracket A \rrbracket = 0$
halt			Halt execution of the RAM.
init	A		Initialize the register at address $\llbracket A \rrbracket$ to zero. (Do nothing if register is already initialized.)
cnst	B	A	$\llbracket A \rrbracket \leftarrow B$
add	A_2	A_1	$\llbracket A_1 \rrbracket \leftarrow \llbracket A_1 \rrbracket + \llbracket A_2 \rrbracket$
sub	A_2	A_1	$\llbracket A_1 \rrbracket \leftarrow \llbracket A_1 \rrbracket - \llbracket A_2 \rrbracket$
mul	A_2	A_1	$\llbracket A_1 \rrbracket \leftarrow \llbracket A_1 \rrbracket * \llbracket A_2 \rrbracket$
div	A_2	A_1	$\llbracket A_1 \rrbracket \leftarrow \llbracket A_1 \rrbracket / \llbracket A_2 \rrbracket$ if $\llbracket A_2 \rrbracket \neq 0$
mov	A_2	A_1	$\llbracket A_1 \rrbracket \leftarrow \llbracket A_2 \rrbracket$
imv	A_2	A_1	$\llbracket A_1 \rrbracket \leftarrow \llbracket \llbracket A_2 \rrbracket \rrbracket$
mvi	A_2	A_1	$\llbracket \llbracket A_1 \rrbracket \rrbracket \leftarrow \llbracket A_2 \rrbracket$

fetches the instruction and performs the corresponding operations.² This involves setting the program counter to the next instruction, by default the successor of the current address. Table I lists the instructions supported by the standard RAM machine. The Peano-arithmetic RAM machine uses a subset of these instructions.

Definition 3.2. 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 I).

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. However, pure CHR without built-in constraints does not provide any arithmetic functionality either. Hence, arithmetic must be encoded, for example using Peano arithmetic.

Definition 3.3. A *standard RAM machine* (or standard-arithmetic RAM) consists of a program and a working memory as described above. The program instruction set is given in Table I.

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.

²If 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.

3.3 CHR machines

We introduce a model of computation to investigate the computability and complexity properties of Constraint Handling Rules, which we call the *CHR machine*.

Definition 3.4. A *CHR machine* is a tuple $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{W})$. The host language \mathcal{H} defines a built-in constraint theory $\mathcal{D}_{\mathcal{H}}$, \mathcal{P} is a CHR program, and $\mathcal{W} \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ is a set of *valid goals*, such that \mathcal{P} is a $\Delta_{\omega_t}^{\mathcal{H}}$ -deterministic CHR program for input \mathcal{W} . The machine takes an input query $\mathbb{G} \in \mathcal{W}$ and executes a derivation $d \in \Delta_{\omega_t}^{\mathcal{H}}|_{\mathbb{G}}$.

Terminology. If the derivation d for \mathbb{G} is finite, we say the machine *terminates* 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 . We will use the following notation: $\mathcal{M}_c(\mathbb{G}) = \mathbb{S}$ is the set of *output constraints*, $\mathcal{M}_b(\mathbb{G}) = \mathbb{B}$ is the *output built-in store*, and $\text{deriv}_{\mathcal{M}}(\mathbb{G}) = d$ is the *derivation* for \mathbb{G} . The machine *accepts* the input \mathbb{G} if d is a successful derivation and *rejects* \mathbb{G} if d is a failed derivation ($\mathcal{M}_b(\mathbb{G}) = \text{fail}$). If d is an infinite derivation, we say the machine *does not terminate*. A *CHR(X) machine* is a CHR machine for which the host language $\mathcal{H} = X$. We use Φ to denote *no host language*: the built-in constraint theory \mathcal{D}_{Φ} 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 Prolog-like variables (that cannot be bound) and (zero-arity) atoms. A *CHR-only machine* is a CHR(Φ) machine.

Definition 3.5. 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 host languages are sufficiently strong. Prolog for instance allows arithmetic using the one-way built-in `is/2`. In CHR program listings where the host language is assumed to be sufficiently strong, we will use a slightly abbreviated notation. For example, if `c/1` is a CHR constraint, we write expressions like “`c(N+1)`”: a host language independent notation that is equivalent to “`M is N+1`, `c(M)`” for CHR(Prolog), to “`c(intUtil.add(N,1))`” for CHR(Java), etc.

4. COMPUTATIONAL POWER AND COMPLEXITY OF CHR MACHINES

A model of computation is called *Turing-complete* if it has the same computational power as Turing machines: every Turing Machine can be simulated in the model and every program of the model can be simulated on a Turing machine.

First (subsection 4.1) we show that the CHR-only machine has at least the computational power of the Turing machine. 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. The second requirement for Turing completeness (a Turing machine which can simulate any CHR machine) is discussed in Section 5. In subsection 4.2 we define time and space complexity functions for all three models of computation. Finally (subsection 4.3) we investigate the complexity of CHR machines that simulate RAM machines.

$$\begin{aligned}
 r_1 @ \mathbf{delta}(Q_i, S_i, Q_o, S_o, \mathbf{left}), \mathbf{adj}(LC, C) \setminus \mathbf{state}(Q_i), \mathbf{cell}(C, S_i), \mathbf{head}(C) \\
 \iff LC \setminus == \mathbf{null} \quad | \quad \mathbf{state}(Q_o), \mathbf{cell}(C, S_o), \mathbf{head}(LC). \\
 r_2 @ \mathbf{delta}(Q_i, S_i, Q_o, S_o, \mathbf{right}), \mathbf{adj}(C, RC) \setminus \mathbf{state}(Q_i), \mathbf{cell}(C, S_i), \mathbf{head}(C) \\
 \iff RC \setminus == \mathbf{null} \quad | \quad \mathbf{state}(Q_o), \mathbf{cell}(C, S_o), \mathbf{head}(RC). \\
 r_3 @ \mathbf{delta}(Q_i, S_i, Q_o, S_o, \mathbf{left}) \setminus \mathbf{adj}(\mathbf{null}, C), \mathbf{state}(Q_i), \mathbf{cell}(C, S_i), \mathbf{head}(C) \\
 \iff \mathbf{cell}(LC, b), \mathbf{adj}(\mathbf{null}, LC), \mathbf{adj}(LC, C), \mathbf{state}(Q_o), \mathbf{cell}(C, S_o), \mathbf{head}(LC). \\
 r_4 @ \mathbf{delta}(Q_i, S_i, Q_o, S_o, \mathbf{right}) \setminus \mathbf{adj}(C, \mathbf{null}), \mathbf{state}(Q_i), \mathbf{cell}(C, S_i), \mathbf{head}(C) \\
 \iff \mathbf{cell}(RC, b), \mathbf{adj}(C, RC), \mathbf{adj}(RC, \mathbf{null}), \mathbf{state}(Q_o), \mathbf{cell}(C, S_o), \mathbf{head}(RC). \\
 \mathit{fail} @ \mathbf{nodelta}(Q_i, S_i), \mathbf{rejecting}(Q_i), \mathbf{state}(Q_i), \mathbf{cell}(C, S_i), \mathbf{head}(C) \implies \mathit{fail}.
 \end{aligned}$$

Fig. 5. The CHR program TMSIM, a Turing machine simulator.

4.1 Computational power of CHR machines

Consider the CHR program TMSIM shown in Figure 5 and the corresponding CHR-only machine $\mathcal{M}_{\text{TM}} = (\Phi, \text{TMSIM}, \mathcal{V}_{\text{TM}})$ (we postpone the definition for \mathcal{V}_{TM} for a while). The program TMSIM simulates Turing machines.

Intuitively, the meaning of the constraints in the TMSIM program is as follows:

- delta/5** encodes the partial transition function δ (the Turing machine program) in the obvious way: the first two arguments correspond to inputs and the last three to outputs;
- nodelta/2** encodes the domain on which δ is undefined;
- rejecting/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 $\mathbf{adj}(A, B)$ 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.

Formally, 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 $I = [i_1, \dots, i_n]$,

$$\text{tm_to_chr}(M, I) = \text{prog_to_chr}(M) \cup \{\mathbf{state}(q_0), \mathbf{head}(c_1)\} \cup \text{tape_to_chr}(I)$$

where `prog_to_chr`(M) is defined as follows:

$$\begin{aligned} \text{prog_to_chr}(M) = & \{\mathbf{delta}(q, s, q^*, s^*, d) \mid (q, s) \in Q \times \Sigma \text{ and } \delta(q, s) = (q^*, s^*, d)\} \\ & \cup \{\mathbf{nodelta}(q, s) \mid (q, s) \in Q \times \Sigma \text{ and } \delta(q, s) \text{ is undefined}\} \\ & \cup \{\mathbf{rejecting}(q) \mid q \in Q \setminus F\} \end{aligned}$$

and `tape_to_chr`(I) = $\bigcup_{j=1}^n \{\mathbf{cell}(c_j, i_j), \mathbf{adj}(c_j, c_{j+1})\} \cup \{\mathbf{adj}(\mathbf{null}, c_1)\}$ where $c_{n+1} = \mathbf{null}$ and the other c_j are unique cell identifiers. Clearly, `tm_to_chr` can be computed in $O(n + |Q \times \Sigma|)$ time — linear in the size of the input tape plus the size of the domain of the Turing machine program. We now define $\mathcal{V}_{\mathbb{T}\mathbb{M}}$ as follows:

$$\mathcal{V}_{\mathbb{T}\mathbb{M}} = \{\text{tm_to_chr}(M, I) \mid M \text{ is a Turing machine for which } I \text{ is an input tape}\}.$$

LEMMA 4.1. $\mathcal{M}_{\mathbb{T}\mathbb{M}}$ is indeed a CHR machine, that is, TMSIM is a $\Delta_{\omega_t}^\Phi$ -deterministic CHR program for input $\mathcal{V}_{\mathbb{T}\mathbb{M}}$.

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. \square

We define a function `chr_to_tm` which returns a Turing machine execution state, given an execution state for a CHR machine: $\text{chr_to_tm} : \Sigma^{\text{CHR}} \rightarrow \Sigma^{\mathbb{T}\mathbb{M}}$:
 $\text{chr_to_tm}(\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle q, P, s, N \rangle$ where q, P, S , and N are such that $\mathbb{G} \uplus \mathbb{S} = \text{prog_to_chr}(M) \cup \{\mathbf{state}(q), \mathbf{head}(c), \mathbf{cell}(c, s)\} \cup \text{tape_to_chr}(\text{tape}(\langle q, P, s, N \rangle))$.
 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 $\mathcal{M}_{\mathbb{T}\mathbb{M}}$ can simulate every Turing machine:

$$\begin{array}{ccc} \text{Turing machine } M, \text{ input } I : \sigma_i^{\mathbb{T}\mathbb{M}} & \xrightarrow{*} & \sigma_f^{\mathbb{T}\mathbb{M}} \\ \downarrow \text{tm_to_chr} & & \uparrow \text{chr_to_tm} \\ \text{CHR machine } \mathcal{M}_{\mathbb{T}\mathbb{M}}, \text{ goal } \mathbb{G} : \sigma_i & \rightsquigarrow_{\Delta_{\omega_t}^\Phi(\text{TMSIM})} & \sigma_f \end{array}$$

THEOREM 4.2. For any Turing machine M and input tape I :

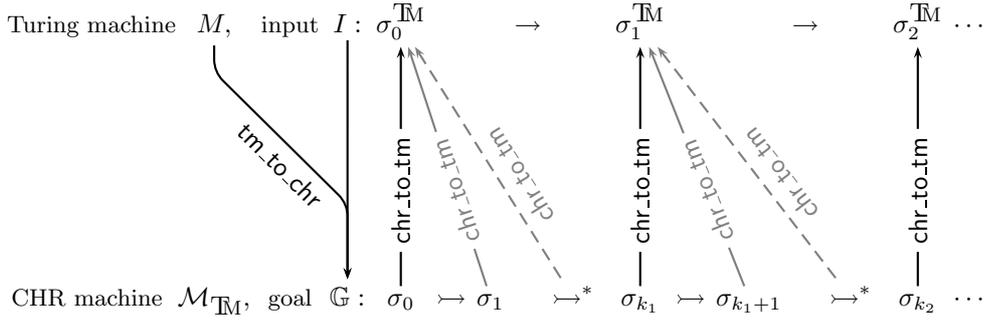
- (1) M terminates on input $I \Leftrightarrow \mathcal{M}_{\text{TM}}$ terminates on input $\text{tm_to_chr}(M, I)$
- (2) M accepts $I \Leftrightarrow \mathcal{M}_{\text{TM}}$ accepts $\text{tm_to_chr}(M, I)$
- (3) M outputs $\text{chr_to_tm}(X)$ on input $I \Leftrightarrow \mathcal{M}_{\text{TM}}$ outputs X on input $\text{tm_to_chr}(M, I)$

PROOF. Let $\sigma_0 = \text{initstate}(\text{tm_to_chr}(M, I))$ be the initial state of \mathcal{M}_{TM} with input $\text{tm_to_chr}(M, I)$. Note that $\text{chr_to_tm}(\sigma_0) = \langle q_0, [], i_1, [i_2, \dots, i_n] \rangle$ is the initial execution state of M .

Observation 1. For every \mathcal{M}_{TM} transition $\sigma_i \rightarrow \sigma_{i+1}$, either

- (a) $\text{chr_to_tm}(\sigma_i) \rightarrow \text{chr_to_tm}(\sigma_{i+1})$ is a TM transition for M , or
- (b) $\text{chr_to_tm}(\sigma_i) = \text{chr_to_tm}(\sigma_{i+1})$.

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 $\sigma_i \rightarrow \sigma_{i+1}$ by the **Apply** transition rule, where the applied rule is r_x . The second case (b) holds if $\sigma_i \rightarrow \sigma_{i+1}$ by the **Introduce** transition rule. Case (b) also holds when $\sigma_i \rightarrow \sigma_{i+1}$ 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. Schematically, the relation between the Turing machine derivation steps and the CHR machine steps is as follows:



Observation 2. If σ_i is a final state (for \mathcal{M}_{TM}), then $\text{chr_to_tm}(\sigma_i)$ is a final state (for M). Furthermore, if σ_i is a successful final state, then $\text{chr_to_tm}(\sigma_i)$ is an accepting final state, and if σ_i is a failure state, then $\text{chr_to_tm}(\sigma_i)$ is non-accepting.

Observation 3. The sub-derivations corresponding to one Turing machine step are finite. In other words, all chains of the form $\text{chr_to_tm}(\sigma_n) = \text{chr_to_tm}(\sigma_{n+1}) = \text{chr_to_tm}(\sigma_{n+2}) = \dots$ 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 corresponding \mathcal{M}_{TM} derivation on input $\text{tm_to_chr}(M, I)$ is also finite. This follows from observation 1 and 3.

The three properties follow straightforwardly from the above observations. \square

4.2 Definitions of time and space complexity

We have shown that any Turing-computable function can be computed in CHR. To investigate complexity properties of CHR machines, we first define the notions of time and space complexity for the three models of computation.

4.2.1 Turing machines

Definition 4.3. The (worst-case) *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:

$$\mathbb{T}TIME_M(n) = \max\{\#deriv_M(I) \mid I \in \Sigma^* \text{ and } |I| = n\}$$

Definition 4.4. The (worst-case) *space complexity* of a Turing machine M is the maximal tape size used in derivations for inputs of a given size:

$$\mathbb{T}SPACE_M(n) = \max\{\#tape(\sigma) \mid \sigma \in deriv_M(I) \text{ and } I \in \Sigma^* \text{ and } |I| = n\}$$

4.2.2 RAM machines

Definition 4.5. The (worst-case) time complexity $\text{RAMTIME}_P(n)$ of a RAM machine with program P is the maximal time needed for an execution starting 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 time which is logarithmic in the absolute value of the numbers involved in the arithmetic operation.

Definition 4.6. The (worst-case) 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 $0, \dots, \text{maxaddr}$ of the number of bits needed to represent the maximal value that it held. Unused registers within that range are charged for one bit for the implicit value 0 stored there.

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.

This definition corresponds to that of Savitch [1978]. In the literature, other definitions of space complexity are often used, that do not count the registers that were unused.³ According to such definitions, a program that uses for input size n (only) the registers with address $2^1, 2^2, 2^3, \dots, 2^n$ has a space complexity of only $\mathcal{O}(n)$ — according to our definition it needs exponential space. An advantage of such definitions is that programs never use more space than time. However, in our opinion the definition of [Savitch 1978] is more realistic.

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.

RAM machines are Turing-complete. Both RAM machines can simulate a T -time Turing machine in $\mathcal{O}(T)$ time. 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 $\mathcal{O}(ST \log^2 S)$ time. The

³See also [van Emde Boas 1990], page 28, for a discussion of different space complexity measures.

A	B	C	D	E	% horizontal:	vertical:
F		G		H	word (A,B,C,D,E),	word (A,F,I),
I	J	K		L	word (I,J,K),	word (J,M,R),
	M	N	O	P	word (M,N,O,P),	word (C,G,K,N,S),
Q	R	S	T	U	word (Q,R,S,T,U),	word (O,T),
					⇒	word (E,H,L,P,U)
					solution (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U).	

Fig. 6. Solver program for some crossword puzzle.

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 $\mathcal{O}((T \log T \log \log T)^2)$ time. Simulating a multi-tape TM on a single-tape TM squares the time complexity [Hartmanis and Stearns 1965], so we have:

LEMMA 4.7. *Any standard RAM machine with time complexity T can be simulated on a Turing machine with time complexity $\tilde{\mathcal{O}}(T^4)$.*

4.2.3 CHR machines

Definition 4.8. Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{G})$, the *time function* $chrtime_{\mathcal{M}}$ returns the derivation length, given a valid goal:

$$chrtime_{\mathcal{M}} : \mathcal{G} \rightarrow \mathbb{N} : \mathbb{G} \mapsto \#deriv_{\mathcal{M}}(\mathbb{G}).$$

Definition 4.9. Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{G})$ and assuming that host language constraints of \mathcal{H} take constant time, the (worst-case) time complexity function $CHRTIME_{\mathcal{M}}$ is defined as follows:

$$CHRTIME_{\mathcal{M}}(n) = \max\{chrtime_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{G} \wedge inputsize(\mathbb{G}) = n\}$$

where *inputsize* is a function which returns the size of a goal.

The time given by $chrtime_{\mathcal{M}}$ is the time needed by a *theoretical* CHR machine. The next section deals with the relation between this theoretical time and practical CHR implementations. This is important because the definition of $chrtime_{\mathcal{M}}$ does not correspond to the reality of CHR implementations, as the following example illustrates.

Example. A CHR machine can solve a crossword puzzle in time linear in the number of words, which does not seem to be achievable on a RAM machine. 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 returns all s solutions for a crossword puzzle in time $\mathcal{O}(n + s)$.

In many cases we simply use the number of conjuncts in the goal \mathbb{G} as its size $inputsize(\mathbb{G})$. In other cases, valid goals consist of just one constraint and the goal size is determined by its arguments.

Example. Consider the CHR machine $\mathcal{M}_P = (\text{Prolog}, \text{PRIMES}, \{\mathbf{upto}(n) \mid n \in \mathbb{N}\})$ (cfr. Figure 2 on page 4). An appropriate goal size function is given by $\text{inputsize}(\mathbf{upto}(n)) = n$. Every derivation starting with $\mathbf{upto}(n)$ consists of $n - 1$ **Solve** steps (one for every **is/2** built-in constraint), $2n - 1$ **Introduce** steps (n **upto/1** constraints and $n - 1$ **prime/1** constraints), and $n + \text{nonprimes}(n) < 2n$ **Apply** steps, where $\text{nonprimes}(n)$ is the number of composite numbers between 2 and n (*loop* is applied $n - 1$ times, *stop* once, and *absorb nonprimes*(n) times). As a result, the time complexity $\text{CHRTIME}_{\mathcal{M}_P}(n) < 5n - 2$ is $\mathcal{O}(n)$.

Definition 4.10 State size function.

$$\text{SIZE} : \Sigma^{\text{CHR}} \rightarrow \mathbb{N} : \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto \text{size}(\mathbb{G}) + \text{size}(\mathbb{S}) + \text{size}(\mathbb{B}) + \text{size}(\mathbb{T})$$

where for sets X , $\text{size}(X) = \sum_{x \in X} |x|$ and the size $|x|$ is the usual term size.

Definition 4.11. Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{V}\mathcal{G})$, the *space function* $\text{chrspac}_{\mathcal{M}}$ returns the worst-case execution state size, given an initial goal:

$$\text{chrspac}_{\mathcal{M}}(\mathbb{G}) = \max\{\text{SIZE}(\sigma) \mid \sigma \in \text{deriv}_{\mathcal{M}}(\mathbb{G})\}$$

Definition 4.12. Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{V}\mathcal{G})$, the (worst-case) space complexity function $\text{CHRSPACE}_{\mathcal{M}}$ is defined as follows:

$$\text{CHRSPACE}_{\mathcal{M}}(n) = \max\{\text{chrspac}_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{V}\mathcal{G} \wedge \text{inputsize}(\mathbb{G}) = n\}$$

Example. Consider, as in the previous example, the CHR machine \mathcal{M}_P and the goal size function $\text{inputsize}(\mathbf{upto}(n)) = n$. If we assume all integers can be represented in a fixed number of bits, the space complexity $\text{CHRSPACE}_{\mathcal{M}_P}$ is $\mathcal{O}(n)$. 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.

4.3 Complexity of CHR machines

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 $\text{CHR}(\mathcal{H})$ machines can simulate standard RAM machines with the same time and space complexity if \mathcal{H} is a sufficiently strong host language.

THEOREM 4.13. *A CHR-only machine $\mathcal{M}_{\text{PARAM}}$ exists which can simulate, in $\mathcal{O}(T + P + I)$ time and $\mathcal{O}(P + J)$ space, a T -time, S -space Peano-arithmetical 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 4.2. Consider the CHR-only program **PARAMSIM**, listed in Figure 7, and the corresponding CHR-only machine $(\Phi, \text{PARAMSIM}, \mathcal{V}\mathcal{G}_{\text{PARAM}})$.

$$\begin{array}{lll}
 \mathbf{i}(L, L_o, \mathbf{inc}, A) \setminus & \mathbf{m}(A, X), \mathbf{c}(L) & \iff \mathbf{m}(A, Z), \mathbf{s}(Z, X), \mathbf{c}(L_o). \\
 \mathbf{i}(L, L_o, \mathbf{dec}, A) \setminus & \mathbf{m}(A, X), \mathbf{s}(X, Y), \mathbf{c}(L) & \iff \mathbf{m}(A, Y), \mathbf{c}(L_o). \\
 \mathbf{i}(L, L_o, \mathbf{clr}, A) \setminus & \mathbf{m}(A, X), \mathbf{c}(L) & \iff \mathbf{m}(A, \mathbf{zero}), \mathbf{c}(L_o). \\
 \mathbf{i}(L, L_o, \mathbf{jmp}, A) \setminus & \mathbf{c}(L) & \iff \mathbf{c}(A). \\
 \mathbf{i}(L, L_o, \mathbf{cjmp}, A, J), & \mathbf{m}(A, \mathbf{zero}) \setminus \mathbf{c}(L) & \iff \mathbf{c}(J). \\
 \mathbf{i}(L, L_o, \mathbf{cjmp}, A, J), & \mathbf{m}(A, X), \mathbf{s}(X, _) \setminus \mathbf{c}(L) & \iff \mathbf{c}(L_o). \\
 \mathbf{i}(L, L_o, \mathbf{halt}) \setminus & \mathbf{c}(L) & \iff \mathbf{true}.
 \end{array}$$

Fig. 7. The CHR program PARAMSIM, a simulator of Peano-arithmetic RAM machines.

The mapping from CHR states to RAM states is as follows. Memory cells are represented as $\mathbf{m}(A, V)$ constraints, where A is the address and V refers to the value. If V is the atom \mathbf{zero} , the value is zero. Otherwise, there is a successor constraint $\mathbf{s}(V, W)$ which expresses that $V = W + 1$. Again, W can be either \mathbf{zero} or there is another successor constraint $\mathbf{s}(W, X)$, etc. For example, to represent that the register r_1 contains the value 3, the following conjunction of constraints could be used: $\mathbf{m}(r_1, N_3), \mathbf{s}(N_3, N_2), \mathbf{s}(N_2, N_1), \mathbf{s}(N_1, \mathbf{zero})$.

The mapping from a RAM program and input to an initial CHR goal is now obvious. We define $\mathcal{W}_{\text{PARAM}}$ to be the image of this mapping. The RAM program is encoded as $\mathbf{i}/\{3,4,5\}$ 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 $\mathbf{c}/1$ constraint.

After the initialization, one rule is applied for every step of the RAM. 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 steps, plus P steps to **Introduce** the $\mathbf{i}/\{3,4,5\}$ constraints. Initializing the input memory cells to their values means inserting one $\mathbf{m}/2$ constraint for every input and I $\mathbf{s}/2$ constraints. The number of registers (and hence $\mathbf{m}/2$ constraints) is fixed for a given program (remember we don't have indirection), so the total time complexity of the CHR machine is $\mathcal{O}(T + P + I)$. The CHR store contains P $\mathbf{i}/\{3,4,5\}$ constraints; since the number of $\mathbf{m}/2$ constraints is fixed and the number of $\mathbf{s}/2$ constraints is bounded by J , the space complexity of the CHR machine is $\mathcal{O}(P + J)$. \square

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 $\mathcal{O}(P + S)$ space by directly storing the value of a register in the second argument of the $\mathbf{m}/2$ 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 4.14. *For any sufficiently strong host language \mathcal{H} , a $\text{CHR}(\mathcal{H})$ machine \mathcal{M}_{RAM} exists which can simulate, in $\mathcal{O}(T + P + S)$ time and $\mathcal{O}(S + P)$ space, a T -time, S -space standard RAM machine with a program of P lines.*

PROOF. As in Theorem 4.13, for the simulator program of Figure 8. The RAM

$i(L, \mathbf{init}, A),$	$\mathbf{m}(A, B), \mathbf{maxm}(M) \setminus \mathbf{c}(L)$	$\iff \mathbf{initm}(M+1, B, L).$
$\mathbf{initm}(A, B, L)$	$\iff A \leq B \mid \mathbf{m}(A, 0), \mathbf{initm}(A+1, B, L).$	
$\mathbf{initm}(A, B, L)$	$\iff A > B \mid \mathbf{maxm}(B), \mathbf{c}(L+1).$	
$i(L, \mathbf{cnst}, B, A) \setminus$	$\mathbf{m}(A, X), \mathbf{c}(L)$	$\iff \mathbf{m}(A, B), \mathbf{c}(L+1).$
$i(L, \mathbf{add}, B, A),$	$\mathbf{m}(B, Y) \setminus \mathbf{m}(A, X), \mathbf{c}(L)$	$\iff \mathbf{m}(A, X+Y), \mathbf{c}(L+1).$
$i(L, \mathbf{sub}, B, A),$	$\mathbf{m}(B, Y) \setminus \mathbf{m}(A, X), \mathbf{c}(L)$	$\iff \mathbf{m}(A, X-Y), \mathbf{c}(L+1).$
$i(L, \mathbf{mul}, B, A),$	$\mathbf{m}(B, Y) \setminus \mathbf{m}(A, X), \mathbf{c}(L)$	$\iff \mathbf{m}(A, X*Y), \mathbf{c}(L+1).$
$i(L, \mathbf{div}, B, A),$	$\mathbf{m}(B, Y) \setminus \mathbf{m}(A, X), \mathbf{c}(L)$	$\iff \mathbf{m}(A, X/Y), \mathbf{c}(L+1).$
$i(L, \mathbf{mov}, B, A),$	$\mathbf{m}(B, Y) \setminus \mathbf{m}(A, X), \mathbf{c}(L)$	$\iff \mathbf{m}(A, Y), \mathbf{c}(L+1).$
$i(L, \mathbf{imv}, B, A),$	$\mathbf{m}(B, C), \mathbf{m}(C, Y) \setminus \mathbf{m}(A, X), \mathbf{c}(L)$	$\iff \mathbf{m}(A, Y), \mathbf{c}(L+1).$
$i(L, \mathbf{mvi}, B, A),$	$\mathbf{m}(B, Y), \mathbf{m}(A, C) \setminus \mathbf{m}(C, X), \mathbf{c}(L)$	$\iff \mathbf{m}(C, Y), \mathbf{c}(L+1).$
$i(L, \mathbf{jmp}, A) \setminus$	$\mathbf{c}(L)$	$\iff \mathbf{c}(A).$
$i(L, \mathbf{cjmp}, A, J),$	$\mathbf{m}(A, 0) \setminus \mathbf{c}(L)$	$\iff \mathbf{c}(J).$
$i(L, \mathbf{cjmp}, A, J),$	$\mathbf{m}(A, X) \setminus \mathbf{c}(L)$	$\iff X \neq 0 \mid \mathbf{c}(L+1).$
$i(L, \mathbf{halt}) \setminus$	$\mathbf{c}(L)$	$\iff \mathbf{true}.$

Fig. 8. The CHR program SRAMSIM, a simulator of standard RAM machines.

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 $0, \dots, m$ and a constraint $\mathbf{maxm}(m)$ is added to the initial CHR goal. The auxiliary constraint $\mathbf{maxm}(m)$ indicates that the current highest initialized register address is m . When a register with a higher address n is initialized, the auxiliary constraint $\mathbf{initm}/3$ is used to initialize all addresses in the range $m+1, \dots, n$. In addition to the $\mathcal{O}(P)$ time to **Introduce** the encoded program and the $\mathcal{O}(T)$ time to simulate it (again one CHR rule application per program instruction), the simulator needs $\mathcal{O}(S)$ time to initialize memory. \square

Note that for a given, fixed RAM program which uses at least as much time as space, P is a constant and S is $\mathcal{O}(T)$, so the CHR simulator SRAMSIM takes $\mathcal{O}(T)$ time and $\mathcal{O}(S)$ space.

5. IMPLEMENTING CHR MACHINES ON RAM MACHINES

The previous section dealt with the complexity properties of the theoretical CHR machine. In this section we investigate the complexity achievable in practice: on a RAM machine. CHR compilers convert CHR programs to (RAM machine) executable code (subsection 5.1). We prove a general result that gives a bound on the complexity of executing a CHR machine on a RAM machine (subsection 5.2). We then apply it to the RAM machine simulator (subsection 5.3). This allows us to conclude that every algorithm can be implemented efficiently in CHR.

5.1 Compilation of CHR

Holzbaur and Frühwirth [1999] pioneered efficient implementations of Constraint Handling Rules by compilation to host language (Prolog) code. The resulting host language code can be executed on RAM machines by an interpreter or a further compilation step. RAM machines can obviously simulate CHR machines since CHR implementations exist:

LEMMA 5.1. *The RAM machine can simulate the CHR machine.*

Because CHR machines can simulate Turing machines (Theorem 4.2), and Turing machines can simulate CHR machines (Lemma 4.7 and the above), we get the following rather unsurprising result:

COROLLARY 5.2. *The CHR machine is Turing-complete.*

5.2 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. One of the most optimizing CHR compilers currently available is the K.U.Leuven CHR system⁴. It was developed by Schrijvers and Demoen [2004], building on the system developed by Holzbaur and Frühwirth [1999]. For more detail on optimizing CHR compilation, see [Holzbaur et al. 2005; Schrijvers 2005; Duck 2005; Sneyers et al. 2008].

First we give an overview of some notions of optimizing compilation that are crucial for time and space efficiency. Given the refined operational semantics, the join ordering strategy, functional dependencies, and efficient constraint store operations, we define the *dependency rank* of a constraint occurrence, which determines the time complexity of executing a CHR machine. We then discuss memory reuse techniques that have an impact on the space complexity. Finally, we formulate a general complexity meta-theorem.

5.2.1 *The refined operational semantics ω_r .* Like most other CHR implementations, the K.U.Leuven CHR system implements the refined operational semantics ω_r , first formalized by Duck et al. [2004]. Since the ω_r semantics instantiates the abstract ω_t semantics, every derivation performed by the K.U.Leuven CHR system corresponds to a valid ω_t derivation. Note that the reverse is not true: in general, not every ω_t derivation corresponds to a ω_r derivation.

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.

5.2.2 *Join ordering.* CHR compilers implement a strategy to pick the order of partner constraint lookups, called the *join ordering* [Holzbaur et al. 2005]. Given a CHR program \mathcal{P} , the join ordering induces for every occurrence c of \mathcal{P} , an order $\prec_c^{\mathcal{P}}$ on its partners. The join ordering strategy of the K.U.Leuven CHR compiler is based on the algorithm introduced in [Holzbaur et al. 2005].

5.2.3 *Functional dependencies.* Duck and Schrijvers [2005] formally defined the (set semantics) functional dependency property and they also proposed an abstract interpretation based analysis to detect this property. 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. We omit the formal definition, referring to [Duck and Schrijvers 2005]. Functional dependencies can be enforced using simpagation rules. For example, to make sure

⁴The K.U.Leuven CHR website: <http://www.cs.kuleuven.be/~toms/CHR/>

that $\mathbf{c}/4$ has a functional dependency on the combination of its first and third argument, a program could start with the rule “ $\mathbf{c}(A, _, B, _) \setminus \mathbf{c}(A, _, B, _) \iff true$ ”.

5.2.4 Constraint store operations. Naive implementations of CHR use Prolog lists to represent the constraint store. This allows insertion in $\mathcal{O}(1)$ time, but look-up and removal take $\mathcal{O}(n)$ time in the worst case (where n is the number of constraints in the store). Optimizing CHR compilers like the K.U.Leuven CHR system use more advanced data structures to implement the constraint store. The join ordering 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 $\mathcal{O}(1)$ (amortized) time.

5.2.5 Determined partners. To obtain a stronger bound on the complexity of finding suitable partner constraints, we introduce the notions of determined partners and dependency rank. These notions depend both on the power of the static analysis in the CHR compiler and on the dynamic program behavior. The former influences the quality of the join order strategy, while the latter determines the functional dependencies in derivations starting with valid goals $\mathcal{W} \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$.

Definition 5.3. Given a join ordering strategy \prec , a CHR program \mathcal{P} , and a set of valid goals \mathcal{W} , we say an occurrence c is *determined* by an (active) occurrence a if for all execution states σ that occur in a derivation $d \in \Delta_{\omega_t}^{\mathcal{H}}|_{\mathbb{G}}$ for some valid goal $\mathbb{G} \in \mathcal{W}$, the following holds: if state σ allows the **Apply** transition for rule r , 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 $x \prec_a^{\mathcal{P}} c$.

Definition 5.4. The *dependency rank* of an (active) occurrence a is the number of non-determined partner constraints of a .

We first illustrate the above definitions. We assume the join ordering strategy \prec used in the K.U.Leuven CHR system, the program SRAMSIM, and the set of valid goals corresponding to the valid RAM machine instances.

Example. Consider the third rule of SRAMSIM:

$$\mathbf{i}(L, \mathbf{add}, B, A), \mathbf{m}(B, Y) \setminus \mathbf{m}(A, X), \mathbf{c}(L) \iff \mathbf{m}(A, X+Y), \mathbf{c}(L+1).$$

With respect to the active occurrence $\mathbf{c}(L)$, the following join ordering is computed: $\mathbf{c}(L) \prec_{\mathbf{c}(L)}^{\text{SRAMSIM}} \mathbf{i}(L, \mathbf{add}, B, A) \prec_{\mathbf{c}(L)}^{\text{SRAMSIM}} \mathbf{m}(B, Y) \prec_{\mathbf{c}(L)}^{\text{SRAMSIM}} \mathbf{m}(A, X)$. Given the first argument L there can be only one $\mathbf{i}/4$, which means the first arguments of the $\mathbf{m}/2$ constraints are known. Again, for any given first argument, there can be only one $\mathbf{m}/2$. These functional dependencies are not enforced explicitly by simpagation rules; they are implied by the set of valid goals and the rules. Since all partners are determined by $\mathbf{c}(L)$, the dependency rank of $\mathbf{c}(L)$ is zero.

The reader can verify that for all occurrences of $\mathbf{c}/1$ in the rules of SRAMSIM, the dependency rank is zero. In general, the dependency rank is of course not always zero. Also, in one rule, different occurrences may have a different dependency rank.

Example. Consider the following rule:

$$\mathbf{m}(A,B), \mathbf{m}(B,C), \mathbf{m}(D,E), \mathbf{m}(E,F), \mathbf{m}(G,H) \iff \dots$$

With respect to the active occurrence $a = \mathbf{m}(A,B)$, we get the following join ordering: $\mathbf{m}(A, B) \prec_a \mathbf{m}(B, C) \prec_a \mathbf{m}(D, E) \prec_a \mathbf{m}(E, F) \prec_a \mathbf{m}(G, H)$. Assuming that the first argument of $\mathbf{m}/2$ uniquely determines the constraint, the dependency rank of $\mathbf{m}(A,B)$ is two: $\mathbf{m}(D,E)$ and $\mathbf{m}(G,H)$ contribute to the dependency rank. The dependency rank of $\mathbf{m}(B,C)$ is three.

5.2.6 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 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 earlier work [Sneyers et al. 2006b] 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 5.5. *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:

$$\forall x \leq n : M_x \leq \max_{i \leq n} S_i \leq \max_{i \leq n+1} S_i = S_{max}$$

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 $M_{n+1} = S_{n+1} \leq S_{max}$. \square

With an unlimited cache size, no constraint representation ever becomes garbage:

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.

5.2.7 *General complexity result.* We now state the main result of this section:

THEOREM 5.6. *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 K.U.Leuven CHR system compiles \mathcal{P} to Prolog code which has, for the given derivation d , a time complexity $\mathcal{O}(TS^{m+1})$ and a space complexity $\mathcal{O}(S + p)$.*

PROOF. Assume the derivation d consists of s **Solve** steps, i **Introduce** steps, and $a = p + r$ **Apply** steps: $T = s + i + a$. The cost of finding a match for an active occurrence is $\mathcal{O}(S^m)$ 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 $\mathcal{O}(a)$ time plus the time to find a match, which we attribute to the **Introduce** and **Solve** transitions. The **Introduce** steps take $\mathcal{O}(iS^m)$ time: matches are tried for every occurrence of the introduced constraint, and for a fixed CHR program, the number of occurrences is bound by a constant. Every **Solve** step potentially triggers all the $\mathcal{O}(S)$ constraints in the store, so the **Solve** steps may take up to $\mathcal{O}(sS^{m+1})$ time. Since s , i , and a are all $\mathcal{O}(T)$, the total time complexity is $\mathcal{O}(TS^{m+1})$. The $\mathcal{O}(S + p)$ space complexity can be achieved using suspension reuse with unlimited cache size, as shown in Lemma 5.5. \square

THEOREM 5.7. *If in the previous theorem, the CHR program is ground (i.e. all constraint arguments are ground), then $\mathcal{O}(TS^m)$ time complexity can be achieved.*

PROOF. In ground programs constraints are never triggered. This reduces the complexity of a single **Solve** transition to a constant. \square

Formulating this result in terms of CHR machines and RAM machines, we get the following corollary.

COROLLARY 5.8. *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 $\mathcal{O}(TS^m)$ and space complexity $\mathcal{O}(S)$, where m is the maximum dependency rank of the active occurrences in the program of the CHR machine.*

5.3 Complexity of SRAMSIM

Now we use the general result of Corollary 5.8 to analyze the time and space complexity of the RAM simulation (compiler-generated code) of the CHR machine \mathcal{M}_{RAM} , which itself simulates RAM machines.

LEMMA 5.9. *The CHR machine \mathcal{M}_{RAM} with program SRAMSIM, with time complexity T and space complexity S can be simulated on a RAM machine with time complexity $\mathcal{O}(T)$ and space complexity $\mathcal{O}(S)$.*

PROOF. It can be easily verified that the dependency rank for all occurrences of $\mathbf{c}/1$ is zero given the join ordering strategy used in the K.U.Leuven CHR system (cfr. the first example in Section 5.2.5). For the other occurrences this is slightly less straightforward. Valid goals have exactly one $\mathbf{c}/1$ constraint. All rules remove one $\mathbf{c}/1$ 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 $\mathbf{c}/1$ constraint — in other words, $\mathbf{c}/1$ has a set semantics functional dependency on the empty key. If the join order strategy does the lookup of $\mathbf{c}/1$ first, the remaining partners become determined.

Since the dependency rank is zero for all occurrences, applying Corollary 5.8 (with $m = 0$) results in the desired complexities. \square

In practice we will add a rule like “ $\mathbf{c}(_) \iff \text{fail}$ ” at the very end of the SRAMSIM program. In the refined semantics, for valid goals, this rule is never applied. However, it allows the *never stored* optimization [Holzbaur et al. 2005] to deduce that all partners of $\mathbf{c}/1$ are *passive*. This implies that only the dependency ranks of the occurrences of $\mathbf{c}/1$ have to be considered.

We conclude that “everything can be done efficiently in CHR”:

COROLLARY 5.10. *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 K.U.Leuven CHR system, with time and space complexity within a constant from the original complexities.*

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 4.14, a CHR(Prolog) machine \mathcal{M}_{RAM} with program SRAMSIM exists, which simulates that RAM machine in $\mathcal{O}(S)$ space and $\mathcal{O}(T)$ time, since P is a constant and S is $\mathcal{O}(T)$. Now, because of Lemma 5.9, executing SRAMSIM in the K.U.Leuven CHR system also takes $\mathcal{O}(T)$ time and $\mathcal{O}(S)$ space. \square

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.

5.4 Discussion: register initialization

Our definition of the space complexity of a RAM machine (see Section 4.2.2) is based on [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 SRAMSIM of Fig. 8 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.

$$\begin{array}{ll}
\mathbf{i}(L, \mathbf{init}, A), \mathbf{m}(A, B), \mathbf{m}(B, \cdot) \setminus \mathbf{c}(L) & \iff \mathbf{c}(L+1). \\
\mathbf{i}(L, \mathbf{init}, A), \mathbf{m}(A, B) \setminus \mathbf{c}(L) & \iff \mathbf{m}(B, 0), \mathbf{c}(L+1).
\end{array}$$

Fig. 9. Alternative implementation of the **init** instruction, using the ω_r semantics.

However, by relying on the refined operational semantics, we can implement the **init** instruction in a different way — see Fig. 9. 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: it depends on the rule application order enforced by the refined semantics for correctness. Indeed, under ω_r semantics the second rule is applied only if the first rule cannot be applied because of the absence a corresponding **m**/2 constraint. It is not known whether checking for absence of constraint is possible in a confluent way [Van Weert et al. 2006].

6. CONSTANT FACTORS AND EXPERIMENTAL RESULTS

In the previous sections we have shown that every algorithm can be implemented in CHR with the right asymptotic time and space complexity. However, the constant factors hidden behind asymptotic complexities could be huge and completely paralyzing in practice. In this section we investigate these constants experimentally.

6.1 Practical relevance of the general complexity result

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 following Intel assembler code (on the left), which corresponds to the following RAM simulator query (on the right) :

<pre> movl \$1, %eax movl \$1000000, %ecx movl \$0, %edx .L1: addl %eax, %edx subl %eax, %ecx je .L5 jmp .L1 .L5: </pre>	<pre> 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). </pre>
--	--

By translating assembler code to a RAM simulator query, we get a CHR programs with the same asymptotic time and space complexity: both the CHR program and the assembler code take linear time and constant space. Figure 18 lists this and some other examples of RAM programs.

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: the RAM simulator uses about 10 seconds while the assembler program uses 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.

6.2 Experimental evaluation

In the rest of this section we investigate the performance of hand-written CHR programs that implement two classical algorithms in an elegant way. In particular, 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.

6.2.1 Setup. The execution times listed in the tables in this section were obtained as follows. We used GCC version 3.3.5 with the `-O4` option to compile the C programs, hProlog⁵ version 2.4.39-32 with the K.U.Leuven CHR compiler by Schrijvers and Demoen [2004], SICStus Prolog⁶ version 3.12.2 (x86-linux-glibc2.2) with CHR version 2.2 by Holzbaur and Frühwirth [1998], and JCHR⁷ version 1.3.3 by Van Weert et al. [2005], executed in Java 1.5.0_03 with the “server” virtual machine and background compilation disabled (options `-server -Xbatch`). For the C programs, we listed the user cpu time. For the CHR programs in Prolog systems, we measured user cpu time, not including garbage collection time. For the JCHR programs, we measured clock time. We made sure the test machine had a very low load, the Java heap size was set as big as possible while still fitting in RAM, and we took the best time of five runs. We tested on a Pentium 4 machine with 512MB of RAM and a cpu with a clock speed of 1.7GHz (cache size 256KB), running Debian GNU/Linux version 3.1 (sarge) with Linux kernel 2.6.15.

The K.U.Leuven CHR system allows the programmer to specify optional type information for the constraint arguments. This optional information can be declared inside the obligatory constraint declarations. The information is used for optimizing compilation. Table II illustrates the three levels of detail we consider. Detail level 2 corresponds to precise type information, level 1 corresponds to coarser groundness information, level 0 gives no information about the constraint arguments at all. The original CHR system in SICStus Prolog, by Holzbaur and Frühwirth [1998], does not have a mechanism to provide type information. In the Java CHR system, precise type declarations are obligatory since Java is a typed language.

6.2.2 Union-find. The classic union-find algorithm of Tarjan and van Leeuwen [1984] efficiently implements the disjoint set union operation. It is not clear whether

⁵hProlog home page: <http://www.cs.kuleuven.be/~bmd/hProlog/>

⁶SICStus Prolog home page: <http://www.sics.se/sicstus/>

⁷JCHR home page: <http://www.cs.kuleuven.be/~petervw/JCHR/>

Table II. Different levels of type information detail in CHR constraint declarations.

Level	Constraint declaration
0	<code>:- chr_constraint root/2, ~~/2, make/1, union/2, link/2, find/2.</code>
1	<code>:- chr_constraint root(+,+), + ~+ , make(+), union(+,+), link(+,+), find(+,-).</code>
2	<code>:- chr_constraint root(+dense_int,+int), +dense_int ~+int, make(+int), union(+int,+int), link(+int,+int), find(+int,-int).</code>

<code>make(A)</code>		\Leftrightarrow <code>root(A,0).</code>
<code>union(A,B)</code>		\Leftrightarrow <code>find(A,X), find(B,Y), link(X,Y).</code>
<code>A ~+B,</code>	<code>find(A,X)</code>	\Leftrightarrow <code>find(B,X), A ~+X.</code>
<code>root(B,-)</code>	<code>find(B,X)</code>	\Leftrightarrow <code>X=B.</code>
<code>find(-,-)</code>		\Leftrightarrow <code>fail.</code>
<code>link(A,A)</code>		\Leftrightarrow <code>true.</code>
<code>link(A,B), root(A,R), root(B,S)</code>		\Leftrightarrow <code>R ≥ S B ~+A, root(A,max(R,S+1)).</code>
<code>link(B,A), root(A,R), root(B,S)</code>		\Leftrightarrow <code>R ≥ S B ~+A, root(A,max(R,S+1)).</code>
<code>link(-,-)</code>		\Leftrightarrow <code>fail.</code>

Fig. 10. The union-find algorithm in CHR.

Table III. Execution times (in seconds) for different implementations of the union-find algorithm. On n elements, n random unions are followed by n random finds.

n	C	CHR(hProlog)			CHR(SICStus)	CHR(Java)
	2	2	1	0	0	2
1k	< 0.01	0.01	0.02	1.46	2.14	0.01
4k	< 0.01	0.03	0.07	32.94	33.12	0.03
8k	< 0.01	0.06	0.14	112.19	119.21	0.07
16k	< 0.01	0.12	0.27	367.93	465.33	0.14
64k	0.02	0.53	1.17	> 1000	> 1000	1.36
256k	0.24	2.31	5.03			7.70
1m	1.12	9.91	21.60			64.82

this algorithm can be implemented with optimal complexity in other declarative languages like pure (side-effect free) Prolog. Schrijvers and Frühwirth [2006] have shown that CHR(Prolog) allows an elegant implementation which has the optimal almost-linear time complexity. It is listed in Figure 10.

Table III lists the execution times for the CHR program in the different CHR systems. We compare these results against a very efficient C implementation⁸. In order to achieve the optimal complexity, type detail level 1 is needed. The high-level CHR(hProlog) implementation is roughly 10 times slower than the corresponding direct low-level implementation in C with type detail level 2. The results are plotted in Figure 11.

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 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

⁸Written by Ariel Faigon, based on a version by Robert Sedgewick. The source code is available at <http://www.yendor.com/programming/minauto/ufind.c>

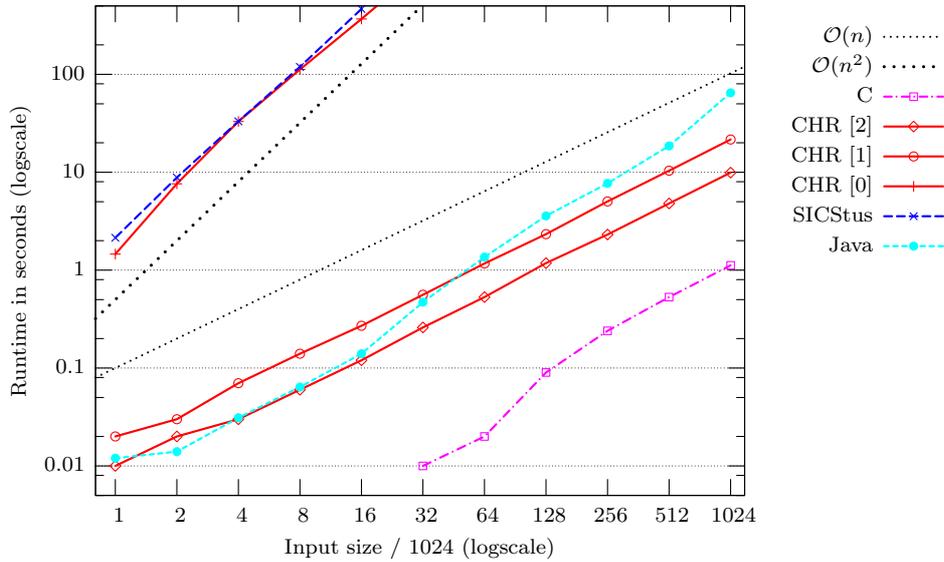


Fig. 11. Plot of the results of the “Union-find” benchmark (cfr. Table III).

 Table IV. Execution times (in seconds) for different implementations of Dijkstra’s algorithm with Fibonacci heaps, on random sparse graphs with n nodes and $4n$ edges.

n	C	CHR(hProlog)			CHR(SICStus)	CHR(Java)
	2	2	1	0	0	2
1k	< 0.01	0.05	0.13	3.47	6.42	0.17
4k	0.01	0.26	0.57	66.24	96.89	0.65
8k	0.02	0.50	1.13	229.88	373.48	1.53
16k	0.07	1.07	2.33	717.92	> 1000	3.77
64k	0.42	4.56	9.85	> 1000		stack overflow
256k	2.06	19.85	41.12			

(cfr. [Schrijvers 2005]: one word for the wrapper functor, two for the constraint arguments, one for the identifier, three for the state). Hence for the union-find algorithm, the CHR version uses about ten times as much space as the C version.

6.2.3 Dijkstra’s algorithm with Fibonacci heaps. The single-source shortest path algorithm of Dijkstra [1959] can be implemented efficiently using the Fibonacci heaps data structure of Fredman and Tarjan [1987]. An implementation of this algorithm in CHR is described in [Sneyers et al. 2006a], where its performance in various CHR systems is also compared to that of a C implementation⁹ by Cherkassky et al. [1996]. The results are listed in Table IV and plotted in Figure 12.

Note the resemblance between Figure 12 and Figure 11. Again, type detail level 1 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 $\mathcal{O}(n \log n)$ time complexity is achieved.

⁹The source code is available at: <http://www.avglab.com/andrew/soft.html>

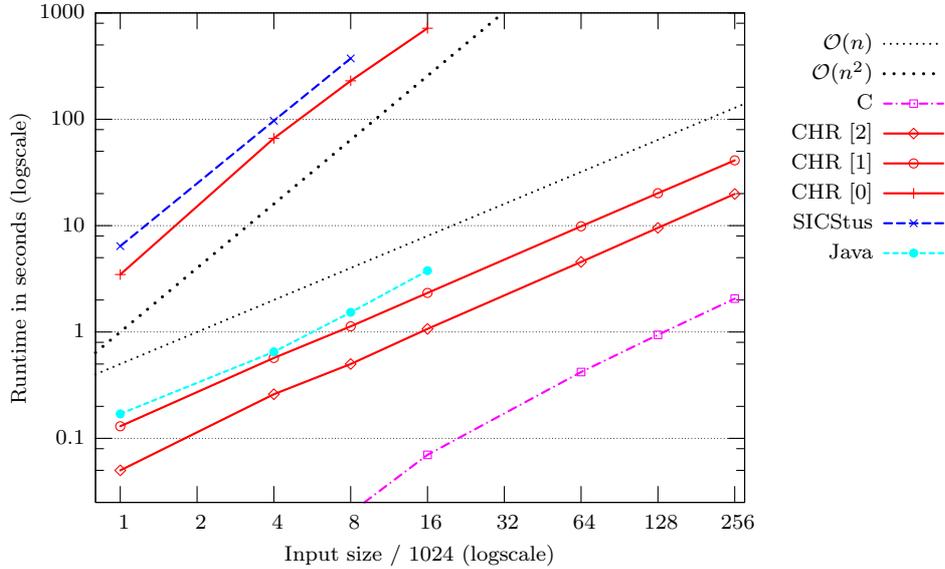


Fig. 12. Plot of the results of the “Dijkstra” benchmark (cfr. Table IV).

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 space usage is about three times more in CHR: for an input size of 256k nodes, the C program uses 23 megabytes, while the CHR(hProlog) program needs 63 megabytes.

6.3 Conclusion: CHR(Prolog) / C \approx 10

The above results indicate that the constant time factor separating CHR(Prolog) 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 sections, we quite confidently state our main thesis:

CONJECTURE 6.1. *The current state of the art in CHR and Prolog systems suffices to implement any algorithm in CHR(Prolog), in a natural and 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 language.*

We expect that a CHR(C) compiler which incorporates state of the art optimizations could push this constant factor significantly closer to 1. The implementation of a CHR(C) compiler is ongoing work [Wuille et al. 2007].

7. SIMILAR RESULTS FOR OTHER LANGUAGES

Among the Turing-complete languages, many have the property we have shown for $X = \text{CHR}(\text{hProlog})$: “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 property is far less trivial. The time and space complexity of a program depends more crucially on an optimizing compiler. Whether or not the general complexity result holds for some language depends largely on the properties of its compiler: a pathological compiler could conceivably detect the pattern of a ‘RAM machine simulator program’ as a special case and produce special, hardwired output with the desired complexity properties.

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 language for some declarative paradigms.

7.1 Sufficient ingredients

An efficient RAM machine simulator can be implemented if the following ingredients can be implemented:

- (1) Iteration which iterates n times in $\mathcal{O}(nT_s)$ time and $\mathcal{O}(S_s)$ space, where T_s and S_s are the time and space needed to evaluate the stop condition of the iteration;
- (2) The arithmetic operations, with the same complexity as the corresponding RAM machine arithmetic operations;
- (3) Constant-time, zero-space *if-then-else* and evaluation of (in)equality conditions (like $X == \text{add}$ and $X \backslash == 0$).
- (4) Growing arrays which allow n insertions, g lookups, and s updates in $\mathcal{O}(n+g+s)$ time and $\mathcal{O}(n)$ space.

Most declarative languages do not offer iteration (the first ingredient) as a basic language construct, but many implementations convert tail recursion to iteration. The second and third ingredient are directly available in the languages we consider.

7.2 Arrays in declarative languages

The remaining ingredient, an efficient dynamically growing array, is the one that seems to be the most difficult to implement.

7.2.1 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 — global variables, mutable terms, *assert/retract* — are allowed, there is a RAM machine simulator implementation: consider, for instance, the code the $\text{CHR}(\text{hProlog})$ compiler generates for SRAMSIM . 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 $\mathcal{O}(\log n)$ worst-case time. Figure 13 lists a Prolog program which uses association lists to implement a RAM simulator. We used the same version of hProlog as the one we used for testing CHR.

Mercury [Somogyi et al. 1996] is a strongly-typed high-performance logic programming language. The Mercury system includes the `array` module, but the procedures in this module are written 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 seems feasible to perform n insertions, g lookups, and s updates in $\mathcal{O}((n + g + s) \log n)$ time and $\mathcal{O}(n)$ space. We have not tested this experimentally since CTGC is not yet available in the main release of the Mercury system.

7.2.2 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 $\mathcal{O}(\min(n, W))$ 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 $\mathcal{O}(n + s)$ instead of $\mathcal{O}(n)$. Figure 14 lists the Haskell program we have tested. We used the Glasgow Haskell Compiler¹⁰ version 6.6 with option “-O2”. We have tested both a “lazy” version (listed in Figure 14) 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 $f e$.

7.2.3 Term-rewrite systems. Maude¹¹ [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]. Figure 15 lists the Maude program we have tested (in Maude version 2.2). 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.

¹⁰GHC home page: <http://www.haskell.org/ghc/>

¹¹Maude home page: <http://maude.cs.uiuc.edu/>

Unfortunately, for the rules listed in Figure 15, 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. It is listed in Figure 16.

7.2.4 Rule engines. Jess¹² [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]. Figure 17 lists the Jess program we have tested (in Jess version 7.0).

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 Figure 15) because the data structure *operations* (`assert`, `retract`, and `modify`) are explicit. Moreover, as far as we know, Jess does not have the join reordering optimization. We have picked the best possible order of rule heads in the simulator program of Figure 17. If the heads are written in a different order, performance will suffer. 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 heads in multi-headed rules.

In general, the eager matching RETE algorithm is asymptotically slower than the lazy matching LEAPS algorithm [Miranker et al. 1990] on which the CHR execution mechanism is based. In the specific case of the RAM machine simulator program of Figure 17, this disadvantage of RETE does not emerge.

7.3 Experimental results

Table V lists the execution times of running the RAM simulator benchmarks of Figure 18 in different RAM simulator implementations. Garbage collection complicates accurate measurement of memory usage, which is why no memory results are given. However, the runtimes include time spent in garbage collection. In this way, space complexity is taken into account. Some benchmarks run out of memory (indicated by “mem”) when the maximal heap size is set to a value slightly lower than the amount of memory available on the test machine (512 Mb).

The first benchmark, “Loop”, performs $\mathcal{O}(n)$ updates on 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 $\mathcal{O}(n)$ lazy thunks so it needs $\mathcal{O}(n)$ 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 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.

In the second benchmark, “MFib”, $\mathcal{O}(n)$ memory cells are used. Naive Maude and CHR without type information do not get the time complexity right: the $\mathcal{O}(n)$ lookups seem to take $\tilde{\mathcal{O}}(n^2)$ time. Prolog, Haskell, and Maude with `Map` get the time complexity almost right: $\tilde{\mathcal{O}}(n)$ instead of $\mathcal{O}(n)$. Both Prolog and Haskell use too much space. In the case of Prolog and strict Haskell, garbage collection takes an

¹²Jess home page: <http://www.jessrules.com/>

```

:- use_module(assoc).
eval(L,Prog,I,0) :- X is L, get_assoc(L,Prog,Instr),
                  (i(Instr,I,I2,X,L1) -> eval(L1,Prog,I2,0) ; 0 = I).
i(add(B,A),I,0,L,L+1) :- get_assoc(B,I,Y), get_assoc(A,I,X,0,Z), Z is X+Y.
i(sub(B,A),I,0,L,L+1) :- get_assoc(B,I,Y), get_assoc(A,I,X,0,Z), Z is X-Y.
...
i(mvi(B,A),I,0,L,L+1) :- get_assoc(B,I,X), get_assoc(A,I,C), get_assoc(C,I,_,0,X).
i(init(A),I,0,L,L+1) :- get_assoc(A,I,B), put_assoc(B,I,0,0).
i(jmp(A),I,I,_,A).
i(cjmp(B,A),I,I,L,L1) :- (get_assoc(B,I,0) -> L1 = A ; L1 is L+1).

```

Fig. 13. A simulator of RAM machines, written in Prolog.

```

module Ramsimul where
import Data.IntMap
data Instr = Add Int Int | Sub Int Int | ... | Cjmp Int Int | Halt
type Program = IntMap Instr
type Memory = IntMap Int
eval :: Int -> Program -> Memory -> Memory
eval pc prog mem =
  case (prog!pc) of
    Add rx ry -> eval (pc + 1) prog (insert ry ((mem!ry) + (mem!rx)) mem)
    Sub rx ry -> eval (pc + 1) prog (insert ry ((mem!ry) - (mem!rx)) mem)
    ...
    Mvi rx ry -> eval (pc + 1) prog (insert (mem!ry) (mem!rx) mem)
    Init rx -> eval (pc + 1) prog (insert (mem!rx) 0 mem)
    Jmp l -> eval l prog mem
    Cjmp r l -> eval (if (mem!r) == 0 then l else pc + 1) prog mem
    Halt -> mem

```

Fig. 14. A simulator of RAM machines, written in Haskell.

```

mod RAMSIMUL is protecting INT .
op c : Int -> State .
op m : Int Int -> State .
op __ : State State -> State [assoc comm] .
ops add sub ... cjmp halt : -> Instr .
rl c(L) i(L,add,B,A) m(B,Y) m(A,X) => c(L + 1) i(L,add,B,A) m(B,Y) m(A,X + Y) .
rl c(L) i(L,sub,B,A) m(B,Y) m(A,X) => c(L + 1) i(L,sub,B,A) m(B,Y) m(A,X - Y) .
...
rl c(L) i(L,init,A) m(A,B) => c(L + 1) i(L,init,A) m(A,B) m(B,0) .
rl c(L) i(L,jmp,A) => c(A) i(L,jmp,A) .
rl c(L) i(L,cjmp,B,A) m(B,0) => c(A) i(L,cjmp,B,A) m(B,0) .
crl c(L) i(L,cjmp,B,A) m(B,Y) => c(L + 1) i(L,cjmp,B,A) m(B,Y) if Y /= 0 .
rl c(L) i(L,halt) => i(L,halt) .
endm

```

Fig. 15. A simulator of RAM machines, written in Maude.

```

mod RAMSIMUL2 is protecting INT .                               protecting MAP{Int,Int}.
op i : Instr Int -> Int .                                       sorts Instr .
op i : Instr Int -> Int .                                       vars B A C L Y X : Int .
op i : Instr -> Int .                                           vars P M : Map{Int,Int} .
ops add sub ... cjmp halt : -> Instr .
op machine : Int Map{Int,Int} Map{Int,Int} -> Map{Int,Int} .
op execute : Int Int Map{Int,Int} Map{Int,Int} -> Map{Int,Int} .
eq machine(L,P,M) = execute(P[L],L,P,M) .
eq execute(i(add,B,A),L,P,M) = machine(L + 1,P,insert(A,(M[A])+(M[B]),M)) .
...
eq execute(i(init,A),L,P,M) = machine(L + 1,P,insert(M[A],0,M)) .
eq execute(i(jmp,A),L,P,M) = machine(A,P,M) .
ceq execute(i(cjmp,B,A),L,P,M) = machine(A,P,M) if (M[B]) = 0 .
eq execute(i(cjmp,B,A),L,P,M) = machine(L + 1,P,M) [otherwise] .
eq execute(i(halt),L,P,M) = M .
endm

```

Fig. 16. A more efficient RAM machine simulator in Maude, using explicit lookups and updates.

```

(deftemplate c (slot at))
(deftemplate m (slot at) (slot val))
(deftemplate i2 (slot at) (slot instr))
(deftemplate i3 (slot at) (slot instr) (slot op))
(deftemplate i4 (slot at) (slot instr) (slot op1) (slot op2))
(defrule add
  ?C <- (c (at ?L))                (i4 (at ?L) (instr add) (op1 ?B) (op2 ?A))
  ?M <- (m (at ?A) (val ?X))        (m (at ?B) (val ?Y))
  => (modify ?M (val (+ ?X ?Y)))    (modify ?C (at (+ ?L 1)))
)
...
(defrule ini
  ?C <- (c (at ?L))                (i3 (at ?L) (instr init) (op ?A))
  (m (at ?A) (val ?B))
  => (assert (m (at ?B) (val 0)))  (modify ?C (at (+ ?L 1)))
)
(defrule j
  ?C <- (c (at ?L))                (i3 (at ?L) (instr jmp) (op ?A))
  => (modify ?C (at ?A))
)
(defrule do_cjump
  ?C <- (c (at ?L))                (i4 (at ?L) (instr cjmp) (op1 ?R) (op2 ?A))
  (m (at ?R) (val 0))
  => (modify ?C (at ?A))
)
(defrule no_cjump
  ?C <- (c (at ?L))                (i4 (at ?L) (instr cjmp) (op1 ?R) (op2 ?A))
  (m (at ?R) (val ~0))
  => (modify ?C (at (+ ?L 1)))
)
(defrule halt
  ?C <- (c (at ?L))                (i2 (at ?L) (instr halt))
  => (retract ?C)
)

```

Fig. 17. A simulator of RAM machines, written in Jess.

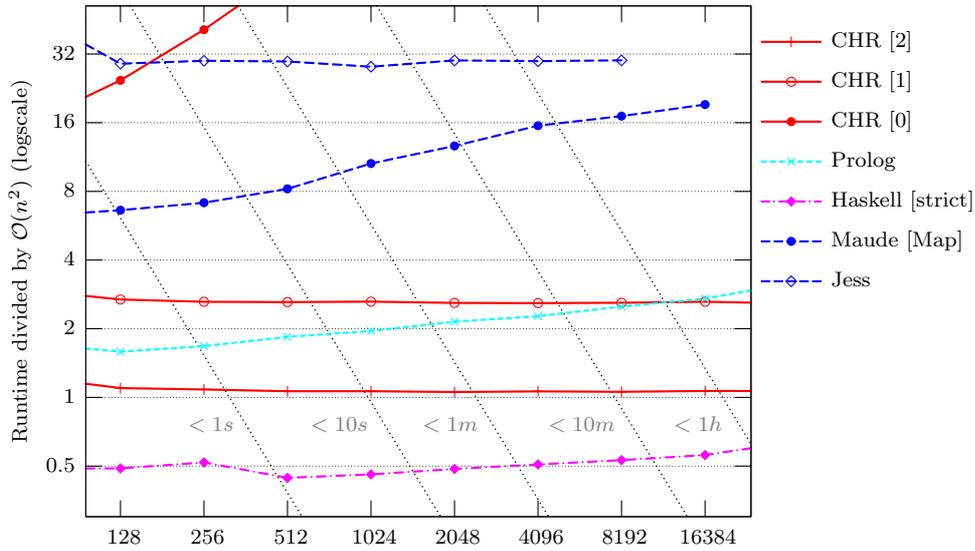


Fig. 19. Plot of the results of the “NLoop” benchmark (cfr. Table V).

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.

In summary:

- CHR without type declarations and naive Maude have the optimal space complexity but they do not achieve the optimal time complexity;
- CHR with type detail level 1 or 2 achieves the optimal time and space complexity;
- Jess achieves the optimal time and space complexity, with a large constant factor;
- Prolog and strict Haskell have a time complexity which is within a polylogarithmic factor from optimal, their space complexity is not optimal;
- lazy Haskell does not get close to the optimal space complexity;
- Maude with `Map` has the optimal space complexity (as far as we can tell) and gets within a (large) polylogarithmic factor from the optimal time complexity.

8. DISCUSSION AND CONCLUSION

We have investigated the computational power and 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-complete 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 K.U.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. We have provided empirical evidence that (at least some) algorithms have a natural

encoding in CHR, which is only ten times slower than a direct C implementation. As far as we know, CHR is the first declarative language for which the “optimal complexity” result can be demonstrated within the pure part of the language, i.e. without imperative extensions to the language.

8.1 Related work

Earlier work by Frühwirth [2002; 2001] studied the time complexity of simplification rules, for naive implementations of CHR. In this approach, a suitable termination order (called a *tight ranking*) is used as an upper bound on the derivation length. Combined with a worst-case estimate of the number and cost of rule application attempts in a naive implementation, this results in a complexity meta-theorem which gives a rough upper bound of the time complexity. For the RAM simulator program SRAMSIM simulating a T -time RAM machine, the upper bound predicted by Frühwirth [2002] is $\mathcal{O}(T^6)$ — quite far from the $\mathcal{O}(T)$ bound of Lemma 5.9.

We have explicitly decoupled the two steps in the approach of Frühwirth [2002] by introducing the notion of a theoretical CHR machine. 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 complexity of the RAM simulating CHR machine is the same as the complexity of the simulated RAM program. The second step corresponds to the question of how efficiently a CHR machine can be executed in practice (i.e. on a RAM machine). Recent work on optimizing compilation of CHR [Schrijvers 2005; Duck 2005] has allowed us to achieve much tighter bounds.

Ganzinger and McAllester [2002] have introduced an abstract logic programming model of computation for which they have studied the time complexity properties. Their formalism is somewhat related to CHR extended with rule priorities, although their notion of deletion is different. It is not clear whether all RAM machine programs can be implemented in their formalism with the same time and space complexity. Experimental results are not available since there is no practical implementation of their formalism. An implementation is currently being developed, which works by translating the formalism to CHR with priorities.

8.2 Future work

The TMSIM program (Figure 5 on page 11) can also simulate non-deterministic Turing machines. However, it is only $\Delta_{\omega_t}^\Phi$ -deterministic for input that corresponds to a deterministic Turing machine. By dropping the determinism condition in the definition of CHR machines, a non-deterministic version of the CHR machine can be defined. The non-determinism can be “don’t care” (if rule application remains committed-choice) or “don’t know” (for a non-committed choice variant of CHR).

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 at the level of CHR source code.

Preliminary results indicate speedups of up to two orders of magnitude for versions of the RAM simulator that were manually specialized for the input RAM program.

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.

The RAM 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. 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.

Our result implies that only the constant factors of CHR can be further improved. A promising approach to reduce those constants is to write a CHR(C) compiler which incorporates the compiler optimizations currently implemented in the K.U.Leuven CHR(Prolog) system. This is work in progress [Wuille et al. 2007].

ACKNOWLEDGMENTS

The authors thank Gregory J. Duck, Peter Van Weert, Leslie De Koninck, and the anonymous reviewers for their helpful comments. Jon Sneyers is funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). This work is partially based on work that was supported in part by the Research Foundation – Flanders (FWO-Vlaanderen) through projects G.0144.03 and G.0160.02. Tom Schrijvers is a post-doctoral researcher of the Research Foundation – Flanders (FWO-Vlaanderen).

REFERENCES

- ADELSON-VELSKY, G. M. AND LANDIS, E. M. 1962. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR* 146, 263–266.
- AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1975. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman, Boston, MA, USA.
- CHERKASSKY, B. V., GOLDBERG, A. V., AND RADZIK, T. 1996. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming* 73, 129–174.
- CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., ET AL. 2002. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science* 285, 2, 187–243.
- CLOCKSIN, W. F. AND MELLISH, C. S. 1984. *Programming in Prolog*. Springer.
- DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 4, 269–271.
- DUCK, G. J. 2005. *Compilation of Constraint Handling Rules*. Ph.D. thesis, University of Melbourne, Victoria, Australia.
- DUCK, G. J. AND SCHRIJVERS, T. 2005. Accurate functional dependency analysis for Constraint Handling Rules. In *CHR '05: Proc. 2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Sitges, Spain, 109–124.
- DUCK, G. J., STUCKEY, P. J., GARCÍA DE LA BANDA, M., AND HOLZBAUR, C. 2004. The refined operational semantics of Constraint Handling Rules. In *ICLP '04: Proc. 20th Intl. Conf. Logic Programming*, B. Demoen and V. Lifschitz, Eds. LNCS, vol. 3132. Springer, Saint-Malo, France, 90–104.
- DUCK, G. J., STUCKEY, P. J., AND SULZMANN, M. 2006. Observable confluence for Constraint Handling Rules. In *CHR '06: Proc. 3rd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Venice, Italy, 61–76.

- EKER, S. 2003. Associative-commutative rewriting on large terms. In *RTA '03: Rewriting Techniques and Applications*, R. Nieuwenhuis, Ed. LNCS, vol. 2706. Springer, Valencia, Spain, 14–29.
- FORGY, C. L. 1982. Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence* 19, 1, 17–37.
- FREDMAN, M. L. AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3, 596–615.
- FRIEDMAN-HILL, E. 2003. *Jess in Action: Java Rule-Based Systems*. Manning Publications.
- FRÜHWIRTH, T. 1992. Constraint simplification rules. Tech. Rep. ECRC-92-18, European Computer-Industry Research Centre, Munich, Germany. July.
- FRÜHWIRTH, T. 1998. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming* 37, 1–3 (Oct.), 95–138.
- FRÜHWIRTH, T. 2001. As time goes by II: More automatic complexity analysis of concurrent rule programs. In *QAPL '01: Quantitative Aspects of Programming Languages*, A. D. Pierro and H. Wiklicky, Eds. Electronic Notes in Theoretical Computer Science, vol. 59(3). Elsevier, Firenze, Italy, 185–206.
- FRÜHWIRTH, T. 2002. As time goes by: Automatic complexity analysis of concurrent rule programs. In *KR '02: Proc. 8th Intl. Conf. Principles of Knowledge Representation and Reasoning*, D. Fensel, F. Guinchiglia, D. McGuinness, and M.-A. Williams, Eds. Morgan Kaufmann, Toulouse, France, 547–557.
- FRÜHWIRTH, T. 2008. *Constraint Handling Rules*. Cambridge University Press. To appear.
- FRÜHWIRTH, T. AND ABDENNADHER, S. 2003. *Essentials of Constraint Programming*. Cognitive Technologies. Springer, New York, NY, USA.
- GANZINGER, H. AND MCALLESTER, D. A. 2002. Logical algorithms. In *ICLP '02: Proc. 18th Intl. Conf. Logic Programming*. LNCS, vol. 2401. Springer, Copenhagen, Denmark, 209–223.
- GIARRATANO, J. C. AND RILEY, G. 1994. *Expert Systems: Principles and Programming*. PWS Publishing Co., Boston, MA, USA.
- HARTMANIS, J. AND STEARNS, R. E. 1965. On the computational complexity of algorithms. *Trans. American Mathematical Society* 117, 285–306.
- HOLZBAUR, C. AND FRÜHWIRTH, T. 1998. CHR reference manual. Tech. Rep. TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria.
- HOLZBAUR, C. AND FRÜHWIRTH, T. 1999. Compiling Constraint Handling Rules into Prolog with attributed variables. In *PPDP '99: Proc. 1st Intl. Conf. Principles and Practice of Declarative Programming*, G. Nadathur, Ed. LNCS, vol. 1702. Springer, Paris, France, 117–133.
- HOLZBAUR, C., GARCÍA DE LA BANDA, M., STUCKEY, P. J., AND DUCK, G. J. 2005. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules 5*, Issue 4 & 5, 503–531.
- HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. 2001. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman, Boston, MA, USA.
- HUDAK, P., HUGHES, J., JONES, S. P., AND WADLER, P. 2007. A history of Haskell: Being lazy with class. In *HOPPL-III: Proc. 3rd ACM SIGPLAN Conf. History of Programming Languages*. ACM Press, San Diego, CA, USA, 1–55.
- MARRIOTT, K. AND STUCKEY, P. J. 1998. *Programming with Constraints: an Introduction*. MIT Press, Cambridge, MA, USA.
- MAZUR, N. 2004. Compile-time garbage collection for the declarative language Mercury. Ph.D. thesis, K.U.Leuven, Leuven, Belgium.
- MIRANKER, D. P., BRANT, D. A., LOFASO, B., AND GADBOIS, D. 1990. On the performance of lazy matching in production systems. In *AAAI '90: Proc. 8th National Conf. Artificial Intelligence*, T. Dietterich and W. Swartout, Eds. MIT Press, Boston, MA, USA, 685–692.
- MORRISON, D. R. 1968. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4, 514–534.
- OKASAKI, C. AND GILL, A. 1998. Fast mergeable integer maps. In *Workshop on ML*. Baltimore, MD, USA, 77–86.

- SAVAGE, J. E. 1998. *Models of Computation*. Addison-Wesley Longman, Boston, MA, USA.
- SAVITCH, W. J. 1978. The influence of the machine model on computational complexity. In *Interfaces between Computer Science and Operations Research*, J. Lenstra, A. R. Kan, and P. van Emde Boas, Eds. Mathematical Centre Tracts, vol. 99. Centre for Mathematics and Computer Science, Amsterdam, 1–32.
- SCHRIJVERS, T. 2005. Analyses, optimizations and extensions of Constraint Handling Rules. Ph.D. thesis, K.U.Leuven, Leuven, Belgium.
- SCHRIJVERS, T. AND DEMOEN, B. 2004. The K.U.Leuven CHR system: Implementation and application. In *CHR '04: 1st Workshop on Constraint Handling Rules, Selected Contributions*, T. Frühwirth and M. Meister, Eds. Ulm, Germany.
- SCHRIJVERS, T. AND FRÜHWIRTH, T. 2006. Optimal union-find in Constraint Handling Rules. *Theory and Practice of Logic Programming* 6, 1&2.
- SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006a. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *WLP '06: Proc. 20th Workshop on Logic Programming*. Vienna, Austria.
- SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006b. Memory reuse for CHR. In *ICLP '06: Proc. 22nd Intl. Conf. Logic Programming*, S. Etalle and M. Truszczynski, Eds. LNCS, vol. 4079. Springer, Seattle, WA, USA, 72–86.
- SNEYERS, J., VAN WEERT, P., DE KONINCK, L., AND SCHRIJVERS, T. 2008. As time goes by: Constraint Handling Rules — A survey of CHR research between 1998 and 2007. Submitted to *Theory and Practice of Logic Programming*.
- SOMOGYI, Z., HENDERSON, F., AND CONWAY, T. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming* 29, 1-3, 17–64.
- TARJAN, R. E. AND VAN LEEUWEN, J. 1984. Worst-case analysis of set union algorithms. *J. ACM* 31, 2, 245–281.
- TURING, A. M. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* 2, 42, 230–265.
- VAN EMDE BOAS, P. 1990. Machine models and simulations. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, J. van Leeuwen, Ed. Elsevier.
- VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. 2005. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *CHR '05: Proc. 2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. Sitges, Spain, 47–62.
- VAN WEERT, P., SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006. Extending CHR with negation as absence. In *CHR '06: Proc. 3rd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. K.U.Leuven, Dept. Comp. Sc., Technical report CW 452. Venice, Italy, 125–140.
- WUILLE, P., SCHRIJVERS, T., AND DEMOEN, B. 2007. CCHR: the fastest CHR implementation, in C. In *CHR '07: Proc. 4th Workshop on Constraint Handling Rules*, K. Djelloul, G. J. Duck, and M. Sulzmann, Eds. Porto, Portugal, 123–137.