CHR for Imperative Host Languages

Peter Van Weert*, Pieter Wuille, Tom Schrijvers**, and Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
FirstName.LastName@cs.kuleuven.be

Abstract. In this paper, we address the different conceptual and technical difficulties encountered when embedding CHR into an imperative host language. We argue that a tight, natural integration leads to a powerful programming language extension, intuitive to both CHR and imperative programmers. We show how to compile CHR to highly optimized imperative code. To this end, we first review the well-established CHR compilation scheme, and survey the large body of possible optimizations. We then show that this scheme, when used for compilation to imperative target languages, leads to stack overflows. We therefore introduce new optimizations that considerably improve the performance of recursive CHR programs. Rules written using tail calls are even guaranteed to run in constant space. We implemented systems for both Java and C, following the language design principles and compilation scheme presented in this paper, and show that our implementations outperform other state-of-the-art CHR compilers by several orders of magnitude.

1 Introduction

Constraint Handling Rules (CHR) [1,26,27,63] is a high-level programming language extension based on guarded, multi-headed, committed-choice multiset rewrite rules. Originally designed for writing user-defined constraint solvers, CHR has matured as a powerful and elegant general purpose language used in a wide range of application domains.

CHR is usually embedded in a CLP host language, such as Prolog [34,51,53] or HAL [17,36]. Real world, industrial software however is mainly written in imperative or object-oriented programming languages. For many problems, however, declarative approaches are more effective. Applications such as planning and scheduling often lead to special-purpose constraint solvers. These are mostly written in the mainstream language itself because a seamless cooperation with existing components is indispensable. Such ad-hoc constraint solvers are notoriously difficult to maintain, modify and extend.

A multiparadigmatic integration of CHR and mainstream programming languages therefore offers powerful synergetic advantages to the software developer. A user-friendly and efficient CHR system lightens the design and development effort required for application-tailored constraint systems considerably. Adhering

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).
** Post-Doctoral Researcher of the Research Foundation – Flanders (FWO-Vlaanderen).

to common CHR semantics further facilitates the reuse of numerous constraint handlers already written in CHR. A proper embedding of CHR in a mainstream language conversely enables the use of innumerous existing software libraries and components in CHR programs.

In the past decade there has been a renewed interest in the integration and use of the rule-based paradigm, in particular *business rules*. Business rules are a technology derived from *production rules*, and are used extensively in real world applications. CHR, with its well-studied clean operational semantics and efficient implementation techniques presents a valid alternative for these tools. Arguably, for CHR to play a role here, embeddings in mainstream languages are required.

Existing CHR embeddings in imperative languages either lack performance [4,76], or are designed to experiment with specific extensions of CHR [78]. Also, in our opinion, these systems do not provide a sufficiently natural integration of CHR with the imperative host language. Instead of incorporating the specifics of the new host into a combined language, these systems port part of the (C)LP host environment as well. This needlessly enlarges the paradigm shift for the programmers of the imperative host language. We will show that a tighter integration of both worlds leads to a useful and powerful language extension, intuitive to both CHR adepts and imperative programmers.

1.1 Overview and Contributions

Our contributions can be summarized as follows:

- We show how CHR can be integrated effectively with an imperative host language. In Section 3, we first outline the different language design issues faced when integrating these two different paradigms. Next, Section 4 outlines our solution, aimed at a tight and natural integration of both worlds. The approaches taken by related systems are discussed in Section 7.
- In Section 5 we present a compilation scheme from CHR to efficient, optimized imperative host language code. We survey generic optimizations, and show how they can be ported to the imperative setting. We also focus on implementation aspects and optimizations specific to imperative target languages.
- We developed mature and efficient implementations of the proposed language design for both Java and C, available at respectively [71] and [81]. The design of both language extensions is presented in Section 4, and their implementations are evaluated in Section 6.

The focus of this article is thus on the design and implementation of CHR systems for imperative host languages. First, we briefly introduce a generic, host language independent syntax and semantics of CHR. For a gentler introduction to CHR, we refer the reader to [17,26,27,51].

2 Preliminaries: CHR Syntax and Semantics

CHR is embedded in a host language that provides a number of predefined constraints, called *built-in constraints*, and a number of data types. The tradi-

tional host language of CHR is Prolog. Its only built-in constraint is equality over its data types, logical variables and Herbrand terms. Asides from built-in constraints, practical implementations mostly allow arbitrary host language procedures to be called as well. Whilst most CHR systems are embedded in Prolog, efficient implementations also exist for Java, Haskell, and C (see Section 7). In this section we only consider the generic, host language independent syntax and semantics of CHR.

2.1 Syntax and Informal Semantics

A CHR program is called a *CHR handler*. It declares a number of user-defined *CHR constraints* and a sequence of *CHR rules*. The rules determine how the handler's CHR constraints are simplified and propagated. A *constraint*, either built-in or CHR, is written $c(X_1, \ldots, X_n)$. Here c is the constraint's *type*, and the X_i's are the constraint's *arguments*. The arguments are elements of a host language data type. The number of arguments, n, is called the constraint's *arity*, and c is called an n-ary constraint, commonly denoted c/n. For nullary constraints the empty argument list is omitted. Trivial nullary built-in constraints are `true` and `false`. Depending on the system, other symbolic notations can be used to express constraints. Equality for instance is mostly written using an infix notation, that is, '$\texttt{X} = \texttt{Y}$' is used instead of e.g. '$\texttt{eq}(\texttt{X}, \texttt{Y})$'.

There are three kinds of CHR rules ($n, n_g, n_b \geq 1$ and $n \geq r > 1$):

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

The *head* of a CHR rule is a sequence, or conjunction, of CHR constraints '$h_1, \ldots, h_n$'. A rule with n head constraints is called an *n-headed* rule; when $n > 1$ it is a *multi-headed* rule. The conjuncts h_i of the head are called *occurrences*. Both the occurrences in a simplification rule and the occurrences '$h_r, \ldots, h_n$' in a simpagation rule are called *removed occurrences*. All other occurrences are *kept occurrences*. The *body* of a CHR rule is a conjunction of CHR constraints and built-in constraints '$b_1, \ldots, b_{n_b}$'. The part of the rule between the arrow and the body is called the *guard*. It is a conjunction of built-in constraints. The guard '$g_1, \ldots, g_{n_g} \mid$ ' is optional; if omitted, it is considered to be '`true` | '. A rule is optionally preceded by a unique *rule identifier*, followed by the '@' symbol.

Example 1. The program LEQ, see Fig. 1, is a classic example CHR handler. It defines one CHR constraint, a less-than-or-equal constraint, using four CHR rules. All three kinds of rules are present. The constraint arguments are logical variables. The handler uses one built-in constraint, namely equality. If the *antisymmetry* is applied, its body adds a new built-in constraint to the built-in constraint solver provided by the host environment. The body of the *transitivity* rule adds a CHR constraint, which will be handled by the CHR handler itself. The informal operational semantics of the rules is explained below, in Example 2.

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

Fig. 1. The CHR program LEQ, a handler for the less-than-or-equal constraint.

Informal Semantics An execution starts from an initial query: a sequence of constraints, given by the user. The *multiset* of all CHR constraints of a CHR handler is called its *constraint store*. The execution proceeds by applying, or *firing*, the handler's rules. A rule is applicable if there are constraints matching the rule's occurrences present in the constraint store for which the guard condition holds. When no more rules can be applied, the execution stops; the final constraint store is called the *solution*.

Rules modify the constraint store as follows. A simplification rule removes the constraints that matched its head, and replaces them with those in its body. The double arrow indicates that the head is logically equivalent to the body, which justifies the replacement. Often, the body is a simpler, or more canonical form of the head. In propagation rules, the body is a consequence of the head: given the head, the body may be added (if the guard holds). As the body is implied by the head, it is redundant. However, adding redundant constraints may allow more rewriting later on. Simpagation rules are a hybrid between simplification and propagation rules: only the constraints matching its removed occurrences, i.e. those after the backslash, are removed if the rule is applied.

Example 2. The first rule of the LEQ handler of Fig. 1, *reflexivity*, replaces a `leq(X,X)` constraint by the trivial built-in constraint `true`. Operationally, this entails removing this constraint from the constraint store. The *antisymmetry* rule states that `leq(X,Y)` and `leq(Y,X)` are logically equivalent to `X = Y`. When firing this rule, the two constraints matching the left-hand side are removed from the store, after which the built-in equality constraint solver is told that `X` and `Y` are equal. The third rule, *idempotence*, removes redundant copies of the same `leq` constraint. It is necessary to do this explicitly since CHR has a *multiset semantics*: multiple instances of the same constraint can reside in the constraint store at the same time. The last rule, *transitivity*, is a propagation rule that computes the transitive closure of the `leq` relation.

2.2 The Refined Operational Semantics

The operational semantics introduced informally in the previous section corresponds to the so-called *high-level* or *theoretical operational semantics* of CHR [20,26]. In this highly non-deterministic semantics, rules are applied in arbitrary order. Most CHR systems though implement a particular, significantly more deterministic instance of this semantics, called the *refined operational semantics* [20]. This semantics is commonly denoted by ω_r. In ω_r, queries and bodies are

executed *left-to-right*, treating the execution of each constraint as a procedure call. When a CHR constraint is executed, this constraint becomes *active*, and looks for matching rules in a *top-to-bottom* order. If a rule fires, the constraints in its body become active first. Only when these are fully handled, the control returns to the formerly active constraint.

The compilation scheme presented in Section 5 implements ω_r, and its optimizations are often justified by properties of this semantics. A sufficiently detailed introduction to this formalism is therefore warranted. For a more complete discussion, we refer the reader to [20,17].

The ω_r semantics is formulated as a state transition system. Transition rules define the relation between subsequent execution states in a CHR execution. Sets, multisets and sequences (ordered multisets) are defined as usual.

Execution state Formally, an execution state of ω_r is a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The first element, the *execution stack* $\mathbb{A}$, is explained below, in the subsection on ω_r's transition rules. The CHR *constraint store* $\mathbb{S}$ is a set of *identified CHR constraints* that can be matched with the rules. An identified CHR constraint $c\#i$ is a CHR constraint c associated with a unique *constraint identifier* i. We introduce the mapping operators $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets in the obvious manner. The constraint identifier is used to distinguish otherwise identical constraints. This is why, even though $chr(\mathbb{S})$ is a multiset of constraints, $\mathbb{S}$ is indeed a set. The *built-in constraint store* $\mathbb{B}$ is the logical conjunction of all built-in constraints passed to the underlying constraint solvers. The *propagation history* $\mathbb{T}$ is a set of tuples, each recording a sequence of constraint identifiers of the CHR constraints that fired a rule, together with that rule's identifier. Its primary function is to prevent trivial non-termination for propagation rules. The integer counter n, finally, represents the next free constraint identifier.

Notation In the following, we use $+\!\!+$ for *sequence concatenation* and $\sqcup$ for *disjoint set union*[1]. For logical expressions X and Y, $vars(X)$ denotes the set of *free variables*, and $\bar{\exists}_Y(X) \leftrightarrow \exists v_1, \ldots, v_n : X$ with $\{v_1, \ldots, v_n\} = vars(X) \backslash vars(Y)$. A *variable substitution* θ is defined as usual. The expression '$\mathcal{D}_b \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}} \theta(G)$' formally states that modelling the *built-in constraint domain* $\mathcal{D}_b$ (see e.g. [51] for a rigorous definition of constraint domains), the built-in store $\mathbb{B}$ entails the guard G after application of substitution θ. For CHR rules a generic simpagation notation is used: '$H_1 \setminus H_2 \Leftrightarrow G \mid B$'. For propagation rules, H_1 is the empty sequence; for simplification rules H_2 is empty.

Transition Rules The transition rules of ω_r are listed in Fig. 3. Given an initial query Q, the *initial execution state* σ_0 is $\langle Q, \emptyset, true, \emptyset \rangle_1$. Execution proceeds by exhaustively applying transitions to σ_0, until the built-in store is unsatisfiable or no more transitions are applicable.

[1] Let X, Y, and Z be sets, then $X = Y \sqcup Z \leftrightarrow X = Y \cup Z \wedge Y \cap Z = \emptyset$.

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

Fig. 2. The LEQ handler annotated with occurrence numbers.

A central concept in this semantics is the *active constraint*, the top-most element on the execution stack $\mathbb{A}$. Each newly added CHR constraint causes an **Activate** transition, which initiates a sequence of searches for partner constraints to match rule heads. Adding a built-in constraint initiates similar searches for applicable rules: a built-in constraint is passed to the underlying solver in a **Solve** transition, which causes **Reactivate** transitions for all constraints whose arguments might be affected. We say these constaints are *reactivated*. CHR constraints whose arguments are *fixed* are not reactivated, the additional built-in constraint cannot alter the entailment of guards on these arguments; formally:

Definition 1. *A variable v is* fixed *by a conjunction of built-in constraints B, denoted $v \in fixed(B)$, iff $\mathcal{D}_b \models \forall\rho(\bar{\exists}_v(B) \wedge \bar{\exists}_{\rho(v)}\rho(B) \rightarrow v = \rho(v))$ for arbitrary renaming ρ.*

The order in which occurrences are traversed is fixed by ω_r. Each active constraint tries its occurrences in a CHR program in a top-down, right-to-left order. The constraints on the execution stack can therefore become *occurrenced* (in **Activate** and **Reactivate** transitions). An *occurrenced* identified CHR constraint $c\#i{:}j$ indicates that only matches with the j'th occurrence of constraint c are considered when the constraint is active.

Example 3. Fig. 2 shows the LEQ program, with all occurrences annotated with their *occurrence number*. Rules are tried from *top-to-bottom*. In this example, this means simplification is tried prior to propagation. Furthermore, occurrences in the same rule are matched with the active constraint from *right-to-left*, ensuring that the active constraint is removed as soon as possible. Both properties can be essential for an efficient execution.

Each active CHR constraint traverses its different occurrences through a sequence of **Default** transitions, followed by a **Drop** transition. During this traversal all applicable rules are fired in **Propagate** and **Simplify** transitions. As with a procedure, when a rule fires, other constraints (its body) are executed, and execution does not return to the original active constraint until after these calls have finished. The different conjuncts of the body are solved (for built-in constraints) or activated (for CHR constraints) in a left-to-right order.

The approach taken by ω_r thus closely corresponds to the execution of the stack-based programming languages to which CHR is commonly compiled. This is why the semantics feels familiar, and why it allows a natural interleaving with host language code (see Section 4). It is also an important reason why the semantics can be implemented very efficiently (see Section 5).

1. Solve $\langle [b|\mathbb{A}], S_0 \sqcup S_1, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle S_1 +\!\!+ \mathbb{A}, S_0 \sqcup S_1, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ where b is a built-in constraint and $vars(S_0) \subseteq fixed(\mathbb{B})$, the variables fixed by $\mathbb{B}$. This causes all CHR constraints affected by the newly added built-in constraint b to be reconsidered.

---

2. Activate $\langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle [c\#n\!:\!1|\mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_{n+1}$ where c is a CHR constraint (which has not yet been active).

---

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

---

4. Simplify $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup H_3 \sqcup S, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail$
$\langle B +\!\!+ \mathbb{A}, H_1 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ where the j-th occurrence of c is d, an occurrence in a (renamed apart) rule ρ of the form:

$$\rho \ @ \ H_1' \setminus H_2', d, H_3' \Leftrightarrow G \mid B$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_k) = \theta(H_k')$ for $1 \leq k \leq 3$, and $\mathcal{D}_b \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}\theta(G)$. Let $t = (\rho, id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

---

5. Propagate $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup H_3 \sqcup S, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail$
$\langle B +\!\!+ [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ where the j-th occurrence of c is d, an occurrence in a (renamed apart) rule ρ of the form:

$$\rho \ @ \ H_1', d, H_2' \setminus H_3' \Leftrightarrow G \mid B$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_k) = \theta(H_k')$ for $1 \leq k \leq 3$, and $\mathcal{D}_b \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}\theta(G)$. Let $t = (\rho, id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

---

6. Drop $\langle [c\#i\!:\!j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if there is no j-th occurrence of c.

---

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

Fig. 3. The transition rules of the refined operational semantics ω_r.

3 Impedance Mismatch

CHR was originally designed to use a (C)LP language as a host. Integrating it with imperative languages gives rise to particular challenges. Imperative host languages do not provide certain language features used by many CHR programs, such as logical variables, search, and pattern matching (Section 3.1). Conversely, the CHR system must be made compatible with the properties of the imperative host. Unlike Prolog, many imperative languages are statically typed, and allow destructive update (Section 3.2).

3.1 (C)LP Language Features

Logical variables Imperative languages do not provide *logical variables*. No reasoning is possible over imperative variables, unless they have been assigned a value. Many algorithms written in CHR however use constraints over unbound variables, or require two, possibly unbound variables to be asserted equal. The latter feature of (C)LP languages is called *variable aliasing*.

Example 4. The constraints of the LEQ handler in Fig. 1 range over logical variables. The body of the *antisymmetry* rule contains an example of aliasing.

The unavailability of a corresponding feature would limit the usefulness of a CHR system in imperative languages. A logical data type, together with library routines to maintain it, therefore has to be implemented in the host language.

Built-in Constraint Solvers More general than variable aliasing, (C)LP languages provide *built-in constraint solvers* for CHR. Prolog provides only one true built-in constraint, namely equality over Herbrand terms. More powerful CLP systems such as HAL offer multiple types of constraint solvers (see [19]). Imperative languages on the other hand offer no built-in constraint support. To allow high level programming with constraints in CHR guards and bodies, underlying constraint solvers need to be implemented from scratch. We refer to Section 4.2 for more information.

Pattern matching CHR uses pattern matching to find applicable rules. In logical languages, pattern matching is readily available through unification[2], even on elements of compound data structures (Herbrand terms). These matches are referred to as *structural matches*. Imperative hosts typically do not provide a suited language construct to perform pattern matching on its (compound) data types. Of course, it is possible to implement a library for Herbrand terms and their unification in the host language. A natural CHR embedding, however, also allows constraints and pattern matching over native data types of the imperative host. Section 3.2 discusses some complications that arise in this context.

[2] Although CHR's pattern matching (sometimes also referred to as *one-way unification*) is different from unification, it is relatively easy to implement matching using the built-in unification facilities of a typical logical language.

Search To solve non-trivial constraint problems constraint simplification and propagation alone is not always enough. Many constraint solvers also require search. As pure CHR does not provide search, many CHR systems therefore implement CHR$^\vee$ (pronounced "CHR-or"), an extension of CHR with disjunctions in rule bodies [2,6]. The built-in support for chronological backtracking typically offered by Prolog and other (C)LP languages makes the implementation of these disjunctions trivial. Providing search for a CHR system embedded in an imperative host, however, requires an explicit implementation of the choice and backtracking facilities.

We do not address this issue in this article. Earlier work extensively studies the combination of CHR with search [39,79] (see Section 7). There remain however some interesting challenges for future work, as undoing changes made after a choice-point becomes particularly challenging if arbitrary imperative data types and operations are allowed. The only practical solution seems to be a limitation of the host language code used in CHR handlers that need to support search.

3.2 Imperative Language Features

Static Typing Unlike Prolog, many imperative languages are statically typed. A natural implementation of CHR in a typed host language would also support typed constraint arguments, and perform the necessary type checking. Calling arbitrary external host language code is only possible if the CHR argument types have a close correspondence with those of the host language.

Complex Data Types The data types provided by imperative languages are typically much more diverse and complex than those used in logical languages. An effective embedding of CHR should support host language data types as constraint arguments as much as possible.

In Section 3.1 we saw that in (C)LP embeddings, CHR handlers use structural matches to specify the applicability of rules on compound data. Providing structural pattern matching on arbitrary compound data structures provided by imperative languages would require specific syntax, and has certain semantical issues, as discussed in the next three subsections.

Modification Problem Contrary to logical languages, imperative languages allow side effects and destructive update. When executing imperative code, arbitrary values may therefore change. If these values are referred to by CHR guards, these modifications may require the reactivation of one or more constraints. Modifications to a constraint's arguments could also render inconsistent the index data structures used by an efficient CHR implementation (see Section 5). In general it can be very hard or impossible for the CHR handler to know when the content of values has changed. In the production rule literature this is referred to as the *Modified Problem* [13,44,45] (we prefer the term modification problem, as modified problem wrongfully suggests the problem is modified).

Non-monotonicity The traditional specification of CHR and its first order logical reading (see e.g. [26]) assumes monotonicity of the built-in constraints, that is: once a constraint is entailed, it remains entailed. If non-monotonic host-language statements are used in a guard, the corresponding rule no longer has a logical reading. This issue is not exclusive to an imperative host language, but certainly more prominent due to the possibility of destructive updates. A consequence of using imperative data structures as constraint arguments is indeed that, often, these values change non-monotonically. CHR rules that were applicable before, or even rules that have been applied earlier, can thus become inapplicable again by executing host language code. This problem is related to the modification problem, but is more a semantical issue than a practical one.

Behavioral Matches As structural matches over imperative data types are often impractical (see above), guards will test for properties of constraint arguments using procedure calls. This is particularly the case for object-oriented host languages: if constraints range over objects, structural matches are impossible if encapsulation hides the objects' internal structure. Guards are then forced to use public inspector methods instead. Matching of objects using such guards has been coined *behavioral matches* [9]. So, not only can it be difficult to determine when the structure of values changes (the modification problem), it can be difficult to determine which changes affect which guards.

4 Language Design

A CHR system for an imperative host language should aim for an intuitive and familiar look and feel for users of both CHR and the imperative language. This entails a combination of the declarative aspects of CHR — high-level programming in terms of rules and constraints, both built-in and user-defined — with aspects of the host language. As outlined in Section 3, such a combination leads to a number of language design challenges. In this section we outline our view on these issues, and illustrate with two CHR system case studies: one for Java [75] and one for C [82].

4.1 Embedding CHR in an Imperative Host Language

A natural integration of CHR with a host language should allow CHR rules to contain arbitrary host language expressions. For the operational semantics, the refined operational semantics is therefore a good choice (see Section 2.2). The left-to-right execution of guards and bodies is familiar to imperative programmers, and eases the interleaving with imperative host language statements. Moreover, to allow an easy porting of existing CHR solvers, support for the same familiar semantics is at least as important as a similar syntax.

Because calling imperative code typically requires typed arguments, it follows that CHR constraints best range over regular host language data types.

In our opinion, this is also the most natural for imperative programmers. Logical variables and other (C)LP data types, such as finite domain variables or Herbrand terms, can always be encoded as host language data types. The CHR compiler could however provide syntactic sugar for (C)LP data types and built-in constraints to retain CHR's high-level, declarative nature of programming.

Our philosophy is contrary to the one adopted by related systems, such as HCHR, JaCK and DJCHR. As seen in Section 7, these systems limit the data types used in CHR rules to typed logical variables (JaCK) or Herbrand terms (HCHR and DJCHR). Host language data then has to be encoded as logical variables or terms, whereas we propose the opposite: not only is using the host's types is more intuitive to an imperative programmer, it also avoids the performance penalty incurred by constantly encoding and decoding of data when switching between CHR and host language. Partly due to the data type mismatch, some of the aforementioned systems simply do not allow CHR rules to call host language code, or only in a very limited manner (see also Section 7).

The limitations imposed by systems such as JaCK and DJCHR, however, could be motivated by the fact that they need to be able to undo changes made in CHR bodies. This language design choice is reasonable for constraint solvers that require either search or adaptation. Practice shows, however, that even Prolog CHR systems are for general purpose programming (see e.g. [63] for a recent survey of CHR applications). These CHR programs do not always use search or adaptation, and can often be expressed naturally without term encodings.

We therefore focus on providing a tight, natural integration of imperative host language features with CHR. The goal is to facilitate a seemless cooperation with software components written in the host language (see also Section 1). We argue that constraints should therefore range over host language data types, and that arbitrary host language expressions must be allowed in rule guards and bodies.

As seen in Section 3, allowing arbitrary imperative data types and expressions in rule guards leads to the *modification problem*. An important aspect of the interaction between a CHR handler and its host is thus that the CHR handler has to be notified of any relevant modifications to the constrained data values. A first, simple solution is for a CHR handler to provide an operation to reactivate all constraints in its store (see Section 5.2). In Section 5.3, we discuss the performance issues with this solution, and propose several optimizations. Where possible, these notifications should furthermore occur transparently, relieving the programmer of the responsibility of notifying after each change.

4.2 Built-in Constraints and Solvers

In the previous section, we argued that arbitrary host language expressions should be allowed. In this section, we show that it remains worthwhile to consider constraints separately. An important motivation will be that the modification problem can be solved effectively for built-in constraints.

The semantics of CHR assumes an arbitrary underlying constraint system (see Section 2). Imperative languages however offer hardly any built-in constraint support (Section 3). Typically, only asking whether two data values are equal is

supported natively, or asking disequality over certain ordered data types. Solvers for more advanced constraints have to be implemented explicitly.

In any case — whether they either built in the host language itself, or realized as a host language library, or even by another CHR constraint handler (see below) — we call these solvers *built-in constraint solvers*, and their constraints *built-in constraints*. The interaction between a CHR handler and the underlying constraint solvers is well defined (after [19]):

- A built-in constraint solver may provide procedures for *telling* new constraints. Using these procedures, new constraints can be added to the solver's constraint store in bodies of CHR rules and the initial query.
- For constraints that occur in guards, the constraint solver must provide a procedure for *asking* whether the constraint is entailed by its current constraint store or not.
- Thirdly, a built-in constraint solver must alert CHR handlers when changes in their constraint store might cause entailment tests to succeed. The CHR handler then checks whether more rules can be fired. Constraint solvers should relieve the user from the responsibility of notifying the CHR handlers, and notify the CHR solver to only reconsider affected constraints. For efficiency reasons, this is typically solved by adding observers [29] to the constrained variables. This is discussed in more detail in Section 5.3.

Example 5. Reconsider the LEQ handler of Example 1. The handler uses one built-in constraint, namely equality over logical variables. For an imperative host language, this constraint will not be natively supported, but implemented as a library. The *antisymmetry* rule is the only rule that uses the *tell* version of this constraint. All rules though use the *ask* version of this built-in constraint to check whether the equality of certain logical variables is entailed (this is more clearly seen when the rules are rewritten to their Head Normal Form, as introduced in Section 5.2: see Fig. 8 of Example 9). Also, when new built-in constraints are told, e.g. by the *antisymmetry* rule, the entailment of these guards may change, and the necessary `leq/2` constraints must be reactivated.

We do not require all built-in constraints to have both an ask and a tell version. Constraints natively supported by an imperative host language for instance, such as built-in equality and disequality checks, typically only have an ask version. Also, traditionally, built-in constraints implemented by a CHR handler only have a tell version. For a CHR constraint to be used in a guard, it requires both an entailment check, and a mechanism to reactivate constraints when the constraint becomes entailed (as explained above). In [54], an approach to automatic entailment checking is introduced, whilst [21] proposes a programming discipline where the programmer is responsible for specifying the entailment checks. Currently though, no system provides *ask* versions of CHR constraints.

A first reason to distinguish constraints from arbitrary host language code is thus that the modification problem is solved efficiently, and transparently to the user. Built-in constraints can therefore safely be used in guards. A second reason is that a CHR compiler may support specific syntactic sugar to ask and tell these constraints (as assumed in the LEQ handler of the previous example).

Cooperating Constraint Systems Multiple CHR handlers and built-in solvers may need to cooperate to solve problems. CHR handlers can for instance share variables constrained by the same built-in constraint solvers, or one CHR handler can be used as a built-in solver for another CHR handler. When implementing multiple solvers and handlers that have to work together, often the need for global data structures arises. Examples include the data structures required for the implementation of search, or an explicit call stack representation (see Section 5). We therefore group such cooperative constraint components under a single *constraint system*. Only solvers and handlers in the same constraint system are allowed to work together.

4.3 CCHR

CCHR [82] is an integration of CHR with the programming language C [38]. CHR code is embedded into C code by means of a `cchr` block. This block can not only contain CCHR constraint declarations and rule definitions, but also additional data-type definitions and imports of host language symbols. Host language integration is achieved by allowing arbitrary C expressions as guards, and by allowing arbitrary C statements in bodies. Functions to add or reactivate CHR constraints are made available to the host language environment, so they can be called from within C.

Constraint arguments are typed, and can be of any C data type except arrays. Support for logical data types is provided, both in the host language and within CHR blocks. CCHR does not have a concept of built-in constraints as introduced in Section 4.2. All 'ask' requests are simply host-language expressions, and 'tell' constraints are host-language statements, which have to be surrounded by curly brackets. It is however possible to declare macro's providing shorter notations for certain operations, a workaround for C's lack of polymorphism. When a data type is declared as logical, such macro's are generated automatically.

Rules follow the normal Prolog-CHR syntax, yet are delimited by a semicolon instead of a dot. This latter would cause ambiguities since the dot is a C operator.

Example 6. In Figure 4 an example is given how to implement the LEQ handler in CCHR. The first line starts the `cchr` block. The next line declares `log_int_t` as a logical version of the built-in C data type `int`. The third line declares a `leq` constraint that takes two logical integers as argument. The four rules of the LEQ handler look very similar to those of Fig. 1. Equality of logical variables is told using the generated `telleq()` macro.

The `test()` function shows how to interact with the CHR handler from within C. The first line of the function initializes the CHR runtime. The next line creates three `log_int_t` variables (`a`, `b` and `c`), and is followed by a line that adds the three `leq` constraints `leq(a,b)`, `leq(b,c)` and `leq(c,a)`. The next line counts the number of `leq` constraints left in the store. The next four lines assert that no CHR constraints are left, and that all logical variables are equal (in C, if the argument of `assert` evaluates to 0, the program is aborted and a diagnostic error message is printed). The function ends with the destruction of

```
cchr {
    logical log_int_t int;
    constraint leq(log_int_t,log_int_t);

    reflexivity  @ leq(X,X) <=> true;
    antisymmetry @ leq(X,Y), leq(Y,X) <=> {telleq(X,Y);};
    idempotence  @ leq(X,Y) \ leq(X,Y) <=> true;
    transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z);
}

void test(void) {
    cchr_runtime_init();
    log_int_t a=log_int_t_create(), b=log_int_t_create(),
        c=log_int_t_create();
    leq(a,b); leq(b,c); leq(c,a);
    int nLeqs=0; cchr_consloop(j,leq_2,{ nLeqs++; });
    assert(nLeqs==0);
    assert(log_int_t_testeq(a,b));
    assert(log_int_t_testeq(b,c));
    assert(log_int_t_testeq(c,a));
    log_int_t_destruct(a);
    log_int_t_destruct(b);
    log_int_t_destruct(c);
    cchr_runtime_free();
}
```

Fig. 4. The CHR program LEQ implemented in CCHR.

the logical variables used, and the release of all memory structures created by the CCHR runtime.

4.4 The K.U.Leuven JCHR System

This section outlines and illustrates the most important language design choices made for the K.U.Leuven JCHR System [71,75]. For a more detailed description of the language extension we refer to the system's user's manual [72].

A handler declaration in K.U.Leuven JCHR is designed to be very similar to a class declaration in Java. Language features such as `package` and `import` declarations, and the access modifiers `public`, `private` and `protected`, are defined exactly as their Java counterparts [31]. To ease the transition from untyped Prolog to strongly typed Java, we further fully support Java's generic types [10,31]. To the best of our knowledge the K.U.Leuven JCHR system is the first typed CHR-system that adequately deals with polymorphic handlers this way.

A JCHR handler declares one or more constraints. As in CCHR, constraint arguments are typed. In principle, any valid Java-type, including primitive types and generic types, can be used. For each handler, and for each of its declared

```
package examples.leq;

import runtime.Logical;
import runtime.EqualitySolver;

public handler leq<T> {
    public solver EqualitySolver<T> builtin;

    public constraint leq(Logical<T>, Logical<T>) infix =<;

    rules {
        reflexivity  @ X =< X <=> true;
        antisymmetry @ X =< Y, Y =< X <=> X = Y;
        idempotence  @ X =< Y \ X =< Y <=> true;
        transitivity @ X =< Y, Y =< Z ==> X =< Z;
    }
}
```

Fig. 5. The LEQ handler using K.U.Leuven JCHR syntax.

constraints, a corresponding Java class is generated. A handler class contains methods to add non-`private` constraints, and to inspect the constraint store. The latter methods return standard Java `Collection` or `Iterator` objects [67]. The handler class itself also implements the `Collection<Constraint>` interface.

Example 7. Fig. 5 shows a polymorphic K.U.Leuven JCHR implementation of the canonical LEQ example. Note that JCHR allows constraints, both built-in and CHR constraints, to be written using infix notation. Fig. 6 shows how the generated classes are used to solve `leq` constraints over `Integer` objects.

JCHR supports user-defined incremental, built-in constraint solvers. The design follows the principles outlined in Section 4.2. Using annotations, a regular Java type is annotated with meta-data that allows the JCHR compiler to derive which built-in constraints are solved by a solver, and which methods to use for asking and telling these constraints. A JCHR handler has to declare all built-in constraint solvers it uses.

Example 8. The K.U.Leuven JCHR System contains an efficient reference implementation for equality over logical variables. Its interface declaration is shown in Fig. 7. It declares a single `eq` constraint, that can also be written using infix notation. This built-in solver is used in the LEQ example of Fig. 5. The `solver` declaration tells the JCHR compiler to use the `EqualitySolver<T>` interface as a built-in solver. Using the annotations, the JCHR compiler knows to use the `askEqual` method to check the implicit equality guards, and to use the `tellEqual` method in the body of the *antisymmetry* rule. Fig. 6 shows how a built-in constraint solver is used to verify that all JCHR constraints are simplified to built-in equalities after adding three `leq` constraints to the handler.

```
...
EqualitySolver<Integer> builtin = new EqualitySolverImpl<Integer>();
LeqHandler<Integer> handler = new LeqHandler<Integer>(builtin);

Logical<Integer> A = new Logical<Integer>(),
        B = new Logical<Integer>(), C = new Logical<Integer>();

handler.tellLeq(A, B);  // A ≤ B
handler.tellLeq(B, C);  // B ≤ C
handler.tellLeq(C, A);  // C ≤ A

// all CHR constraints are simplified to built-in equalities:
assert handler.getLeqConstraints().size() == 0;
assert builtin.askEqual(A, B);
assert builtin.askEqual(B, C);
assert builtin.askEqual(A, C);
...
```

Fig. 6. A code snippet illustrating how the JCHR LEQ handler and equality built-in solvers are called from Java code.

```
@JCHR_Constraint(identifier = "eq", arity = 2, infix = "=")
public interface EqualitySolver<T> {
    @JCHR_Tells("eq")
    public void tellEqual(Logical<T> X, T val);
    @JCHR_Tells("eq")
    public void tellEqual(T val, Logical<T> X);
    @JCHR_Tells("eq")
    public void tellEqual(Logical<T> X, Logical<T> Y);

    @JCHR_Asks("eq")
    public void askEqual(Logical<T> X, T val);
    @JCHR_Asks("eq")
    public void askEqual(T val, Logical<T> X);
    @JCHR_Asks("eq")
    public void askEqual(Logical<T> X, Logical<T> Y);
}
```

Fig. 7. The declaration of a built-in equality constraint solver interface using annotations.

Next to high-level constraint programming, the K.U.Leuven JCHR also allows arbitrary Java objects and methods to be used. An adequate, efficient solution for the modification problem though, which would allow behavioral matches over arbitrary Java Bean objects [69], is an important part of future work. Interaction with Java already possible though: the user simply needs to reactivate all constraints in case of relevant changes explicitly.

5 Optimized Compilation

Considerable research has been devoted to the efficient compilation and execution of CHR programs, mostly with Prolog as the host language. An early, very influential implementation was the SICStus implementation described in [34]. Its operational semantics was the basis for the refined operational semantics ω_r (see Section 2.2), and its compilation scheme has been adopted by state-of-the-art systems such as HALCHR [17,36] and K.U.Leuven CHR [51,53].

We show how this compilation scheme can be ported to the imperative setting. The structure of this section is similar to that of [51, Chapter 5: *The Implementation of CHR: A Reconstruction*]. Section 5.2 presents a simple compilation scheme for CHR. This naive scheme, whilst obviously correct with respect to ω_r, is fairly inefficient. In Sections 5.3 and 5.4, we gradually transform it into a very efficient CHR compilation scheme. Equivalents of most optimizations discussed in Section 5.3 are also implemented in the Prolog and HAL systems. Section 5.4 however addresses an important technical issue that only arises when adapting the scheme to imperative host languages.

For the compilation scheme presented below, we use imperative pseudo-code. It can easily be instantiated for any concrete imperative language. The instance used by the K.U.Leuven JCHR system to compile CHR to Java is described in detail in [70].

Before the compilation scheme is introduced, Section 5.1 abstractly describes the data structures and the operations it uses. The efficient implementation of these data structures is beyond the scope of this article.

5.1 Basic Data Structures and Operations

The constraint store The main data structure of a CHR handler is the *constraint store*. Each stored CHR constraint is represented as a *constraint suspension* in the constraint store. For each constraint, a constraint suspension data type is generated containing the following fields:

`type` The type of the constraint.
`args` A list of fields containing the constraint's arguments. The type of these arguments is derived from the constraint's declaration.
`id` Each constraint suspension is uniquely identified by a *constraint identifier*, as in the refined operational semantics.
`alive` A boolean field indicating whether the constraint is *alive* or not.

`activated` A boolean indicating whether the constraint has been (re)activated. This field is used for optimizations (see Sections 5.3 and 5.4).

`stored` A boolean field set to `true` if the constraint is stored in the constraint store. Due to the *Late Storage* optimization (Section 5.3), suspensions may represent constraints that are not stored in the constraint store.

`hist` A constraint suspension may contain fields related to the propagation history. More details can be found in Section 5.3.

The constraint suspension may contain further fields, used for instance for constant time removal from the constraint store data structures. These implementation details are beyond the scope of this article though.

In the pseudo-code used throughout this article, a constraint suspension of an n-ary constraint is denoted as $c(X_1, \ldots, X_a)\#\text{ID}$. We assume the transition from a constraint identifier ID to its corresponding constraint suspension can be made, and we often make this transition implicitly. In other words, a constraint identifier is very similar to a pointer to a constraint suspension.

The basic constraint store operations are as follows:

`create(c, [args])` Creates a constraint suspension for a constraint with given type `c` and argument list *args*, and returns its constraint identifier. Because arity and argument types are constraint specific, concrete implementations most likely offer specific `create_c` operations for each constraint type `c`.

`store(ID)` Adds the referenced constraint suspension (created earlier with the `create` operation) to the constraint store.

`reactivateAll` Reactivates all constraints in the store, using the `reactivate(ID)` operation. Optionally, only constraints whose arguments are modifiable are reactivated (as in ω_r's **Solve** transition, see Fig. 3).

`reactivate(ID)` Reactivates the constraint with the given identifier.

`kill(ID)` Removes the identified constraint suspension from the constraint store data structures, and sets its `alive` field to `false`.

`alive(ID)` Tests whether the corresponding constraint is alive or not.

`lookup(c)` Returns an iterator (see below) over all stored constraint suspensions of constraint type c.

To iterate over candidate partner constraints, we use *iterators* [29]. This common abstraction can easily be implemented in any imperative language. Even though probably all CHR implementations rely on some form of iterators, their necessary requirements have never been fixed explicitly. We require the iterators returned by `lookup` operations to have at least the following properties:

robustness The iterators are robust under constraint store modifications. If constraints are added or removed whilst a constraint iteration is suspended, iteration can be resumed from the point where it was suspended.

correctness The iterators only return constraint suspensions that are alive.

completeness All constraints that are stored at the moment of the iterator's creation are returned at least once in the iteration.

weak termination A contiguous iteration does not contain duplicate suspensions. Only if constraint store modifications occur whilst an iteration is suspended, constraints returned prior to this suspension are allowed to be returned once more.

Iterators are preferred to satisfy a stronger termination property, namely **strong termination**, which requires that an iterator returns a constraint suspension at most once.

Iterators offered by predefined data structures typically do not have all required properties. Iterators returned by most standard Java data structures [67], for instance, are not robust under modifications.

The propagation history A second important data structure for any CHR implementation is the *propagation history*. Abstractly, the propagation history contains *tuples*, each containing a rule identifier and a non-empty sequence of constraint identifiers (denoted '`[ID+]`'). We assume the following two operations:

`addToHistory(`*`rule`*`,[ID+])` Adds a tuple to the propagation history.

`notInHistory(`*`rule`*`,[ID+])` Tests whether a given tuple is in the propagation history or not.

5.2 Basic Compilation Scheme

A CHR rule ρ with h occurrences in its head has the following generic form:

$$\rho \mathbin{@} \mathrm{c}_1^{[j_1]}(\mathrm{X}_{1,1}, \ldots, \mathrm{X}_{1,a_1}), \ldots, \mathrm{c}_{r-1}^{[j_{r-1}]}(\mathrm{X}_{r-1,1}, \ldots, \mathrm{X}_{r-1,a_{r-1}}) \setminus \\ \mathrm{c}_r^{[j_r]}(\mathrm{X}_{r,1}, \ldots, \mathrm{X}_{r,a_r}), \ldots, \mathrm{c}_h^{[j_h]}(\mathrm{X}_{h,1}, \ldots, \mathrm{X}_{h,a_h}) \Leftrightarrow \mathrm{g}_1, \ldots, \mathrm{g}_{n_g} \mid \mathrm{b}_1, \ldots, \mathrm{b}_{n_b}.$$

The occurrences in a rule are numbered from left to right, with r the index of the first removed occurrence. For a simplification rule there are no kept occurrences (i.e., $r = 1$), for a propagation rule there are no removed occurrences ($h = r-1$). The *occurrence number* j_i in $\mathrm{c}_i^{[j_i]}$ denotes that this occurrence is the j_i'th occurrence of constraint c_i in the program, when numbered according to the top-to-bottom, right-to-left order determined by ω_r, as defined in Section 2.2.

In this generic form, also called the *Head Normal Form (HNF)*, all arguments $\mathrm{X}_{\alpha,\beta}$ in a rule's head are variables. Moreover, variables never occur more than once in a rule's head, that is: all equality guards implicitly present in the head are written explicitly in the guard.

Example 9. Fig. 8 shows the normalized version of the LEQ handler, with occurrence numbers added for illustration purposes.

Listing 1 shows the compilation scheme for an occurrence $\mathrm{c}_i^{[j_i]}$ in such a rule. Lines 2–7 constitute a nested iteration over all $h - 1$ candidate partner constraints. A rule is applicable on some combination of constraints if all constraints are alive (line 8) and mutually distinct (lines 9–11), and if the guard

```
1  procedure occurrence_c_i_j_i(ID_i, X_{i,1}, ..., X_{i,a_i})
2     foreach c_1(X_{1,1}, ..., X_{1,a_1})#ID_1 in lookup(c_1)
3        ⋱
4           foreach c_{i-1}(X_{i-1,1}, ..., X_{i-1,a_{i-1}})#ID_{i-1} in lookup(c_{i-1})
5              foreach c_{i+1}(X_{i+1,1}, ..., X_{i+1,a_{i+1}})#ID_{i+1} in lookup(c_{i+1})
6                 ⋱
7                    foreach c_h(X_{h,1}, ..., X_{h,a_h})#ID_h in lookup(c_h)
8                       if alive(ID_1) and ... and alive(ID_h)
9                          if ID_1 ≠ ID_2 and ... and ID_1 ≠ ID_h
10                            ⋱
11                               if ID_{h-1} ≠ ID_h
12                                  if g_1 and ... and g_{n_g}
13                                     if notInHistory(ρ, ID_1, ..., ID_h)
14                                        addToHistory(ρ, ID_1, ..., ID_h)
15                                        kill(ID_r)
16                                        ⋮
17                                        kill(ID_h)
18                                        b_1
19                                        ⋮
20                                        b_{n_b}
21                                     end
22                                  end
23                               end
24                            ⋰
25                         end
26                      end
27                   end
28                ⋰
29             end
30          end
31       ⋰
32    end
33 end
```

Listing 1. The compilation scheme for a single occurrence. The active constraint is $c_i(X_{i,1}, \ldots, X_{i,a_i})$, with constraint identifier ID_i.

```
reflexivity  @ leq[1](X, X1) ⇔ X = X1 | true.
antisymmetry @ leq[3](X, Y), leq[2](Y1, X1) ⇔ X = X1, Y = Y1 | X = Y.
idempotence  @ leq[5](X, Y) \ leq[4](X1, Y1) ⇔ X = X1, Y = Y1 | true.
transitivity @ leq[7](X, Y), leq[6](Y1, Z) ⇒ Y = Y1 | leq(X, Z).
```

Fig. 8. The LEQ handler in *Head Normal Form*. Occurrence numbers are added for illustration purposes (as in Fig. 2).

```
1  procedure c(X1,...,Xn)
2     ID = create(c, [X1,...,Xn])
3     store(ID)
4     activate(ID)
5  end
6
7  procedure reactivate(ID)
8     activate(ID)
9  end
10
11 procedure activate(c(X1,...,Xn)#ID)
12    occurrence_c_1(ID,X1,...,Xn)
13    ...
14    occurrence_c_m(ID,X1,...,Xn)
15 end
```

Listing 2. Compilation scheme for an n-ary constraint `c` with m occurrences throughout the program. For each occurrence, lines 4–6 call the corresponding occurrence procedure (see Listing 1).

is satisfied (line 12). After verifying that the rule has not fired before with the same combination of constraints (line 13), the rule is fired: the propagation history is updated (line 14), the constraints that matched removed occurrences are removed from the constraint store (lines 15–17), and the body is executed (lines 18–20).

For each n-ary constraint `c` a procedure `c`$(\texttt{X}_1, \ldots, \texttt{X}_n)$ is then generated by the compiler as depicted in Listing 2. These procedures are used for executing CHR constraints in the body of rules, or for calling CHR from the host language. Also for each constraint `c` an instance of the (polymorphic) procedure `activate`$(c(X_1, \ldots, X_n)\#\texttt{ID})$ is generated. Called by both `c`$(\texttt{X}_1, \ldots, \texttt{X}_n)$ and `reactivate(ID)`, it deals with trying all occurrence procedures in order.

With the basic compilation scheme, it is the responsibility of the built-in constraint solvers to call `reactivateAll` each time a built-in constraint is added. This operation calls the `reactivate(ID)`, also shown in Listing 2, for all constraints in the store (see also Section 5.1). As a simple optimization, constraints without modifyable arguments should not be reactivated, as indicated in the corresponding **Solve** transition of ω_r.

Correctness The basic compilation scheme of Listings 1–2 closely follows the refined operational semantics (see Section 2.2). It is therefore not hard to see it is correct. Lines 1–2 of Listing 2 correspond with an **Activate** transition: the constraint is assigned a constraint identifier and stored in the constraint store. The remaining lines constitute a sequence of **Default** transitions, chaining together the different occurrence procedures.

Each occurrence procedure, as shown in Listing 1, has to perform all applicable **Propagate** or **Simplify** transitions. The body is executed left-to-right as a sequence of host language statements, thus mapping the activation stack onto the host's implicit call stack.

The only subtlety is showing that in a sequence of **Propagate** transitions, all required partner constraint combinations are effectively found by the nested iterations of lines 2–7. The order in which the partners have to be found is not determined by ω_r. The **completeness** and **correctness** properties of the iterators guarantee that an iteration contains at least all constraints that existed at the creation of the iterator, and that are still alive on the moment the iterator is advanced. However, constraints that are added to the store *after* the creation of an iterator, i.e. by an execution of the body, are not required to appear in the iteration. These constraints, however, have been active themselves, so any combination involving them has already been tried or applied. As the propagation history prevents any re-application, not including these constraints in iterations is correct.

Running example The following example will be used as a running example for illustrating the different optimizations throughout the next section:

Example 10. Consider the following rule from the RAM simulator example [59]:

```
add @ mem(B,Y), prog(L,ADD,B,A) \
      mem(A,X), pc(L) <=> mem(A,X+Y), pc(L+1).
```

This rule simulates the ADD instruction of a *Random Access Machine*. The full program can be found in Appendix A. The program of the simulated RAM machine is represented as `prog` constraints. The current program counter `L` is maintained in a `pc` constraint. If `L` refers to an ADD instruction, the above rule is applicable. It looks up two cells of the RAM machine's memory, and replaces one of them with a cell containing the sum of their values, before advancing to the next instruction by adding an incremented program counter.

After HNF transformation, the *add* rule becomes:

```
add @ mem(B,Y), prog(L1,$1,B1,A) \ mem(A1,X), pc(L)
    <=> A = A1, B = B1, L = L1, $1 = ADD | mem(A,X+Y), pc(L+1).
```

The code for the `pc` occurrence in this rule, using the basic compilation scheme, is shown in Listing 3.

5.3 Optimizations

This section describes a number of optimizations for the basic compilation scheme presented in the previous section. Most of these optimizations are not new, and have been applied for compiling CHR to (C)LP as well. Our contribution is a first clear survey that places the many optimizations mentioned in recent literature [14,17,18,36,51,57,62,61,73] in one coherent framework. Even though introduced

```
1  procedure occurrence_pc_1(ID4,L)
2     foreach mem(B,Y)#ID1 in lookup(mem)
3        foreach prog(L1,$1,B1,A)#ID2 in lookup(prog)
4           foreach mem(A1,X)#ID3 in lookup(mem)
5              if alive(ID1) and alive(ID2) and alive(ID3) and alive(ID4)
6                 if ID1 ≠ ID2 and ID1 ≠ ID3 and ID1 ≠ ID4
7                    if ID2 ≠ ID3 and ID2 ≠ ID4
8                       if ID3 ≠ ID4
9                          if A = A1 and B = B1 and L = L1 and $1 = ADD
10                            if notInHistory(add,ID1,ID2,ID3,ID4)
11                               addToHistory(add,ID1,ID2,ID3,ID4)
12                               kill(ID3)
13                               kill(ID4)
14                               mem(A,X+Y)
15                               pc(L+1)
16                            end
17                         end
18                      ...
```

Listing 3. Naive compilation of the `pc(L)` occurrence of the RAM simulator rule.

and illustrated for an imperative host language, the overview provided in this section is useful for any reader interested in optimized compilation of CHR, or any other forward chaining rule-based language. Implementation aspects more specific to imperative target languages are discussed in Section 5.4.

Loop-Invariant Code Motion The tests on lines 9–12 of Listing 1 should be performed as early as possible. Otherwise a phenomenon denoted *trashing* could occur, where tests depending only on outer loops fail for all iterations of the inner loops. So guards are scheduled as soon as all required variables are present[3], and the identifiers of new candidate partner constraints are immediately compared to those of the candidates already found. Only identifiers of constraints of the same type have to be compared.

The `alive` tests on line 8 are not yet moved, since the liveness of partner constraints may change when the rule is committed. Failure to test the liveness of all partners before the next body execution might result in a rule being applied with dead constraints. The optimization of the `alive` tests is addressed later.

Example 11. The optimized compilation of the RAM simulator example introduced in the previous section is listed in Listing 4. Scheduling the '`L = L1`' on line 4 avoids enumerating all `mem(A,X)` memory cells before the right program instruction is found. The search for partner constraints is not yet optimal though. Further optimizations will address several remaining issues.

[3] Note that we assume all guards to be monotonic (see Section 3.2). If a satisfied guard might become unsatisfied by executing a body, scheduling this guard early is not allowed for **Propagate** transitions.

```
1  procedure occurrence_pc_1(ID4,L)
2     foreach mem(B,Y)#ID1 in lookup(mem)
3        foreach prog(L1,$1,B1,A)#ID2 in lookup(prog)
4           if B = B1 and L = L1 and $1 = ADD
5              foreach mem(A1,X)#ID3 in lookup(mem)
6                 if ID1 ≠ ID3
7                    if A = A1
8                       if alive(ID1) and alive(ID2) ... and alive(ID4)
9                          if notInHistory(add,ID1,ID2,ID3,ID4)
10                            ⋮
```

Listing 4. The compilation of the RAM simulator example of Listing 3 after *Loop-Invariant Code Motion*.

Indexing The efficient, selective lookup of candidate partner constraints is indispensable. To achieve this, *indexes* on constraints are used.

Example 12. In Listing 4 of the RAM simulator example, line 3 iterates over all `prog` constraints, each time immediately testing the '`L = L1`' guard. There will however be only one `prog` constraint with the given instruction label `L` (see also the *Set Semantics* optimization). Using an index to retrieve this single constraint, reduces the linear time complexity of this part of the partner constraint search to constant time. A similar reasoning applies to line 5 of Listing 4.

For lookups of partner constraints via known arguments, tree-, hash-, or array-based indexes are used (see e.g. [17,36,51,60]). Tree-based indexes can be used not only for equality-based lookups, but also for pruning the partner constraint search space in case of disequality guards (see [19]). The other two indexing types are particularly interesting as they offer (amortized) constant time constraint store operations. Care must be taken that indexes remain consistent after modifications to the indexed arguments. These techniques are therefore often only used for unmodifiable constraint arguments.

One indexing technique for unbound logical variables commonly used by CHR implementations is *attributed variables* [33,34]. With this technique, variables contain references to all constraints in which they occur. This allows constant time lookups of partner constraints via shared variables.

Imperative host languages naturally allow for direct and efficient implementations of index data structures [75,82]. In fact, performance-critical parts of the hash indexes of the K.U.Leuven CHR system for SWI-Prolog [51,53] have recently been reimplemented in C for efficiency.

Indexes are incorporated into the general compilation scheme by extending the `lookup` operation. The extended operation accepts an additional set of conditions, allowing the combination of a `lookup` with one or more subsequent guards. This operation may make use of existing indexes to obtain all constraints satisfying the requested conditions, or any superset thereof. In the latter case, the conditions not guaranteed by the index are checked within the

```
1 procedure occurrence_pc_1(ID4,L)
2   foreach mem(B,Y)#ID1 in lookup(mem)
3     foreach prog(L1,$1,B1,A)#ID2 in lookup(prog,{B=B1,L=L1,$1=ADD})
4       foreach mem(A1,X)#ID3 in lookup(mem,{A=A1})
5         if ID1 ≠ ID3
6           ⋱
```

Listing 5. Compilation of the RAM simulator's `pc(L)` occurrence. This version improves Listing 4 by incorporating the *Indexing* optimization.

iterator. This way, constraints returned by the iterator are always guaranteed to satisfy the provided guard conditions. By using index lookups that only return the requested constraints, suitable candidate partner constraints are looked up far more efficiently.

Example 13. Listing 5 shows the optimized compilation of our running example. If the specialized `lookup` operations on lines 3–4 use array- or hash-based indexing, both partner constraints are found in $\mathcal{O}(1)$ time. Without indexing, the time complexity is $\mathcal{O}(p \times m)$, with p the number of lines of the RAM program, and m the number memory cells used by the RAM machine.

Join Ordering The time complexity of executing a CHR program is often determined by the *join ordering* — the order in which partner constraints are looked up in order to find matching rules. So far this order was determined by the order they occur in the rule.

The join order determines the earliest position where guards, and thus indexes, may be used. The general principle behind *Join Ordering* is to maximize the usage of indexes, in order to minimize the number of partner constraints tried. The optimal join order may depend on dynamic properties, such as the number of constraints in the store for which certain guards are entailed. Sometimes *functional dependency analysis* [17,18,36] can determine statically that certain indexed lookups return at most one constraint (see also the *Set Semantics* optimization). Without functional dependencies though (or without proper indexing), a compiler must rely on heuristics to determine the join order. The most comprehensive treatment of the join ordering problem is [14].

Example 14. Line 2 of Listing 5 iterates over all `mem` constraints. Lacking any information on `B` and `L`, there is no possibility to use an index using the standard join order. The join order depicted in Listing 6, on the other hand, first looks up the `prog` constraint using the known `L`. Next both `mem` partners are looked up using the known `A` and `B`. In all three cases, if proper indexing is used, only one partner constraint is retrieved (see also Example 15), as the first argument of both the `prog/4` and the `mem/2` constraint are unique identifiers. The latter property may be derived statically from the full RAM program as listed in Appendix A using functional dependency analysis.

```
1  procedure occurrence_pc_1(ID4,L)
2     foreach prog(L1,$1,B1,A)#ID2 in lookup(prog,{L=L1,$1=ADD})
3        foreach mem(A1,X)#ID3 in lookup(mem,{A=A1})
4           foreach mem(B,Y)#ID1 in lookup(mem,{B=B1})
5              if ID1 ≠ ID3
6                 ⋱
```

Listing 6. Compilation of the `pc(L)` occurrence of Listing 5 with optimal *Join Ordering*.

```
1  procedure occurrence_pc_1(ID4,L)
2     prog(L1,$1,B1,A)#ID2 = lookup_single(prog,{L=L1})
3     if ID2 ≠ nil
4        if $1 = ADD
5           mem(A1,X)#ID3 = lookup_single(mem,{A=A1})
6           if ID3 ≠ nil
7              mem(B,Y)#ID1 = lookup_single(mem,{B=B1})
8              if ID1 ≠ nil and ID1 ≠ ID3
9                 ⋱
```

Listing 7. The compilation scheme for the RAM simulator rule occurrence after applying *Set Semantics* to Listing 6.

Set Semantics The functional dependency analysis may show at compile time that a certain lookup will result in at most one constraint [18,36]. In this case, more efficient data structures can be used for the constraint store and its indexes, and specialized lookup routines can be used that return a single suspension instead of an iterator. Such specialized routines are denoted `lookup_single`.

Example 15. In our running example, all lookups have set semantics after applying *Join Ordering* (see Example 14). All loops can thus be turned into simple conditionals, as shown in Listing 7. As an index on `prog`'s first argument alone already yields at most one result, the test on $\$_1$ is placed outside the lookup.

Early Drop and Backjumping As seen in Section 5.1, iterators are guaranteed not to return dead constraints. Constraints may be removed though when matching removed occurrences, or indirectly during the execution of the body. In the naive compilation scheme of Listing 1, this leads to many useless iterations where the active constraint, or certain partner constraints, are no longer alive.

Once the active constraint is killed, we should stop handling it. We call this an *Early Drop*. For this optimization, the `activate` operation of Listing 2 is replaced with the version of Listing 8. Occurrence routines are modified to return a boolean: `true` if trying further occurrences is no longer necessary; `false` otherwise. The alive test for the active constraint is thus removed from line 8 of Listing 1, and replaced with a '**if not** `alive(ID`$_i$`)` **return true**' statement

```
1  procedure activate(c(X_1, ..., X_n)#ID)
2     if occurrence_c_1(ID,X_1,...,X_n) return
3     ...
4     if occurrence_c_m(ID,X_1,...,X_n) return
5  end
```

Listing 8. Compilation scheme for an n-ary constraint `c` with m occurrences throughout the program. This is an updated version of the `activate` procedure of Listing 2, performing an *Early Drop* for the active constraint if required.

right after the body. At the end of the occurrence code finally (i.e., after line 32 of Listing 1), a '**return false**' is added. This is the default case, signifying that any remaining occurrences must still be tested for the current active constraint.

A similar optimization is possible for the partner constraints. When using the scheme of Listing 1, a form of trashing similar to the one seen in *Loop-Invariant Code Motion* may occur. If, for instance, the first partner constraint dies by executing the body, all nested loops are still fully iterated. Since the first partner is already dead, all these lookups and iterations are useless. So, if a constraint dies, we should instead immediately continue with the next constraint for the corresponding loop. The alive tests for the partner constraints are therefore moved after the alive test for the active constraint (i.e., after the body as well). The constraint of the outermost iterator is tested first. If one of the partner constraints tests dead after the execution of the body, a jump is used to resume the corresponding iteration. This optimization, known as *Backjumping*, avoids the form of trashing described above.

All alive tests are now placed after the body instead of before it. This is allowed because at the start of a body execution, each partner was either just returned by an iterator (which guarantees liveness), or tested for liveness after the previous body execution.

In case of a **Simplify** transition, the active constraint is always killed. The *Early Drop* therefore becomes unconditional ('**return true**'), and all further alive become unreachable, and should be omitted. Similarly, removed partner constraints will always be dead after the body. The alive test of the outermost removed partner constraint can therefore be omitted, and replaced with an unconditional backjump. All following alive tests thus becomes redundant. If static analysis shows the active constraint or certain partner constraints cannot be killed during the execution of the body, the corresponding alive tests can be dropped. One trivial case is when the body is empty.

Example 16. In the RAM simulator example, the active `pc` constraint is removed by the rule, so all alive tests can be replaced by a single unconditional '**return true**' after the rule body. See Listing 9.

Non-Robust Iterators Due to the highly dynamic nature of the CHR constraint store, the robustness property of iterators, as specified in Section 5.2, is

hard to implement and often has a considerable performance penalty. There are however cases where this property is not required:

1. If after the execution of a rule body an iterator is never resumed due to an unconditional early drop, or an unconditional backjump over the corresponding loop, introduced by the previous optimization.
2. If static analysis shows the body of a rule is guaranteed not to modify the CHR constraint store.

Robust and non-robust iterators are sometimes called *universal* and *existential* iterators [36,51]. We prefer the term *non-robust iterator*, because they can also be used to iterate over more than one partner constraint (see case 2 above). Non-robust iterators are used where possible because they can typically be implemented more efficiently.

Example 17. In case of the RAM simulator example, all iterators are already superseded by the single-constraint lookups since *Set Semantics* was applied; otherwise, they could have been replaced by non-robust iterators because of the unconditional *Early Drop* in the body.

Late Storage In the default compilation scheme, see Listing 2, constraints are stored immediately after they are told, as in ω_r. A constraint's lifetime, however, is often very short. This is most apparent when the active constraint is removed shortly after activation. The goal of the *late storage* optimization is to postpone adding the constraint to the constraint store as long as possible. In many cases the constraint will then be killed before it is stored. This avoids the considerable overhead of adding and removing the constraint to the constraint store. The performance gain is particularly significant if indexes are used.

During the execution of a body in a **Propagate** transition the formerly active constraint might be required as a partner constraint, or it might have to be reactivated. A straightforward implementation of the optimization therefore stores the active constraint prior to every non-empty body in **Propagate** transitions. To further delay constraint storage the *observation analysis* [51,57] can be used. This static program analysis determines whether a specific body requires a constraint to be stored or not. Finally, if not stored earlier, the active constraint is stored after all occurrence procedures are called (i.e. line 3 is moved after line 4 in Listing 2).

Late Allocation As constraints are not always stored, constraint suspensions do not always have to be created either. Late allocation and late storage are considered separately, because a distributed propagation history maintenance (cf. next optimization) might require allocation earlier than storage. In the optimized compilation schemes of Section 5.4, a constraint suspension may also be allocated earlier if needed as a continuation.

```
1  procedure occurrence_pc_1(ID4,L)
2     prog(L1,$1,B1,A)#ID2 = lookup_single(prog,{L=L1})
3     if ID2 ≠ nil
4        if $1 = ADD
5           mem(A1,X)#ID3 = lookup_single(mem,{A=A1})
6           if ID3 ≠ nil
7              mem(B,Y)#ID1 = lookup_single(mem,{B=B1})
8              if ID1 ≠ nil and ID1 ≠ ID3
9                 kill(ID3)
10                kill(ID4)
11                mem(A,X+Y)
12                pc(L+1)
13                return true
14             end
15          end
16       end
17    end
18    return false
19 end
```

Listing 9. The compilation scheme for the `pc(L)` occurrence after applying *Early Drop* to Listing 7. Also, no propagation history is kept since the occurrence is part of a simpagation rule.

Propagation History Maintenance In the basic compilation scheme, tuples are added to the propagation history, but never removed (line 14 of Listing 1). However, it is obvious that tuples referring to removed constraints are redundant. Tuples added for simplification and simpagation rules, immediately become redundant in lines 15–17, so a propagation history is only kept for propagation rules. The propagation history remains a memory problem nevertheless.

There exist several techniques to overcome this problem. Immediately removing all propagation history tuples a constraint occurs in once it is removed is a first possibility. Practice shows however that this is difficult to implement efficiently. CHR implementations therefore commonly use ad-hoc garbage collection techniques, which in theory could result in excessive memory use, but perform adequately in practice. A first such technique is to remove tuples referring to dead constraints during `notInHistory` checks (see [17]). A second is denoted *distributed propagation history* maintenance, for which suspensions contain propagation history tuples they occur in (see [51]). When a constraint suspension is removed, part of the propagation history is removed as well. These techniques can easily be combined. Other, more advanced garbage collection techniques could be applied as well.

Example 18. The *add* rule of the RAM example is a simpagation rule, so maintaining a propagation history for it is unnecessary. This is reflected in Listing 9.

Propagation History Elimination Despite the above techniques, the maintenance of a propagation history remains expensive, and has a considerable impact on both the space and time performance of a CHR program [73]. For rules that are never matched by reactivated constraints, however, [73] proves that the history can either be eliminated, or replaced by very cheap constraint identifier comparisons. The same paper moreover shows that reapplication is generally more efficient than maintaining a propagation history, and presents a static analysis that determines when rule reapplication has no observable effect. Together, these optimizations cover most propagation rules occurring in practice.

Guard Simplification For each occurrence, guard simplification looks at earlier removed occurrences to infer superfluous conjuncts in the guard. This optimization is best described in [62]. The expected performance gain for guard simplification in itself is limited. Simplifying guards, however, does improve results of other analyses, such as the detection of passive occurrences described in the next optimization.

Passive Occurrences An important goal of several CHR analyses is to detect *passive occurrences*. An occurrence is passive if it can be derived that the corresponding rule can never fire with the active constraint matching it. Detecting passive occurrences is important, not only because superfluous searches for partner constraints are avoided, but also because any index structures only required for these searches do not have to be maintained.

Subsumption analysis Where guard simplification tries to replace guards with `true`, subsumption analysis uses similar techniques to replace guards with `false`. An occurrence is *subsumed* by another occurrence if each constraint that matches the former constraint also matches the latter (taking into account guards and partner constraints). An occurrence that is subsumed by an earlier removed occurrence can be made passive. More information can be found in [62].

Example 19. In the LEQ handler of Fig. 1, the kept occurrence of the *idempotence* rules is clearly subsumed by the removed one (recall from Section 2.2 that ω_r specifies that removed occurrences are tried first). By symmetry, one of the occurrences of the *antisymmetry* rule can be made passive as well (as ω_r considers occurrences from right-to-left, the first occurrence will be made passive).

Never stored analysis If one of the partners of an occurrence is known never to be stored in the constraint store, that occurrence can also be made passive. The basic analysis [17,36] determines that a constraint is never stored if:

- The constraint occurs in a single-headed, guardless simplification rule. The *Guard Simplification* optimization helps considerably by removing redundant guards in these rules.
- The *Late Storage* analysis shows that the constraint is never stored prior to the execution of a body.

Never stored constraints also do not require any constraint store data structures.

Example 20. Given the complete RAM handler program (Appendix A), a CHR compiler can derive that the `pc` constraint is never stored. All other occurrences in the *add* rule of Example 10 are therefore passive. Because most occurrences of `mem` constraints are thus found passive, a compiler can also derive that less indexes have to be built for looking up `prog` constraints.

Selective Constraint Reactivation The naive approach reactivates all suspended constraints (with modifiable arguments) for each modification to a constrained value. This corresponds to the unoptimized **Solve** transition in the refined operational semantics. Reconsidering always *all* constraints though is clearly very inefficient. An obvious optimization is to reactivate only those constraints whose arguments are affected. For an efficient interaction with CHR, constrained data values should therefore maintain references to the CHR constraints they occur in. In terms of the well-known observer pattern [29]: CHR constraints have to observe their arguments.

As explained in Section 4.2, built-in constraint solvers should perform selective reactivation transparently to the user. Changes in a built-in constraint store can typically be reduced to a limited set of variables. This allows the reactivation of only those CHR constraints that are affected by a change in the constraint store. This is analogous to a typical implementation of a constraint solver: constrained variables contain references to all constraints they occur in (see e.g. [48]). When extending logic programming languages with constraint solvers, *attributed variables* are typically used for this [32].

Several further optimizations are possible to avoid more redundant work on reactivation:

- If two unbound (logical) variables are told equal, only the constraints observing one of these variables have to be reactivated. This is correct because all rules that become applicable by telling this equality constraint necessarily contain constraints over both variables.
- So-called *wake conditions* can be used to reconsider only those occurrences whose guard might be affected. This becomes particularly interesting for more complex built-in constraints, such as finite domain constraints. For more information we refer to [19]. Closely related are the *events* used in efficient implementations of constraint propagators [58].

As argued in Sections 3–4, the modification problem should also be addressed for arbitrary host language values. Asides from the `reactivateAll` operation, more selective constraint reactivation needs to be possible. Possible solutions include a well-defined use of the observer pattern, or reactivate operations with user-definable filter functions.

Delay Avoidance By the previous optimization, a constraint is reactivated each time one of its arguments is modified. If the compiler can prove though

that these modifications cannot affect the outcome of a rule's guard, there is no need to reactivate. This is the case if a particular argument does not occur in any guard, or only in anti-monotonous guards (see [52]). A constraint does not have to observe one of its arguments if none of its occurrences has to be reconsidered when it is modified. More details can be found in [52] and [74, Appendix A].

Memory Reuse The memory reuse optimizations of [61] can also be ported to the imperative setting. Two classes of optimizations are distinguished:

- *Suspension reuse*: Memory used by suspensions of removed constraints can be reused for newly added constraints.
- *In-place updates*: In-place updates go one step further. If a constraint is removed and immediately replaced in the rule's body, it is possible to reuse the suspension of the former. This is particularly interesting if the new constraint is of the same constraint type, and only slightly different from the removed constraint. It could then be that the suspension does not have to be removed and re-added to certain indices.

There are subtle issues when implementing this optimization. For more details, we refer to [61].

Example 21. In the RAM simulator example, both constraints added in the body are replacing removed constraints. Using a `replace(ID, indexes, values)` that assigns new values to the arguments on the specified indices, lines 9–13 of Listing 9 can safely be replaced by:

```
⋮
replace(ID3, [2], [X+Y])
replace(ID4, [1], [L+1])
⋮
```

We assume the `replace` operation also activates the updated constraint. In this case, updating these arguments should not require any constraint store operations. The only index on the `mem` constraint for instance is on its first argument. Updating the `X` argument of the $\texttt{mem(A,X)\#ID}_3$ suspension does not require this index to be adjusted.

Drop after Reactivation If a rule fires, the refined operational semantics determines that the active constraint is suspended — i.e., pushed on the activation stack — until the body is completely executed. During the execution of the body this constraint may be reactivated. In this case, when the execution continues with the suspended constraint, all applicable rules matching it have already been tried or fired by this reactivation. Searching for more partner constraints, and continuing with further occurrences, is then superfluous.

Traditionally, this optimization is implemented using an integer field incremented each time the constraint is reactivated (see e.g. [51], which also contains

a correctness proof). Here, we propose a slightly more efficient implementation, which also generalizes better when considering the optimized compilation scheme presented in Section 5.4. We use a boolean field, `activated`, in the constraint suspension, which is set to `true` *after* each reactivation. Prior to a **Propagate** transition, the active constraint's `activated` field is set to `false`. If after the execution of the body, it has become `true`, the constraint must have been reactivated, and the handling of this active constraint can safely be terminated using an early drop (i.e., by returning `true`, as in the *Early Drop* optimization).

This optimization is not applied if static analysis determines that the body never reactivates the active constraint. Obvious instances include when the body is empty, or, if all arguments of the active constraint are unmodifiable. In all other cases, this optimization may save a lot of redundant work.

5.4 Recursion Optimizations

Any non-trivial CHR program contains recursion, i.e., directly or indirectly, there are rules with an occurrence of c/n in the head that activate a body that add a constraint of the same type c/n to the store. In such a case, the compilation schema presented in the previous two sections generates a set of mutually recursive host language procedures. We rely on the host language compiler for generating the final executable, or on the host language interpreter for the eventual execution of our generated code. If the host language does not adequately deal with recursion, the naive compilation scheme leads to stack overflow issues.

Prolog implementations perform tail call optimization since the early days of Prolog. This optimization consists in reusing the execution frame of the caller for the last call in of a clause's body. Prolog thus executes tail calls in constant stack space. For a host language like Prolog, recursion is therefore less of a problem: to solve call stack overflows during the execution of a CHR program it mostly suffices to rewrite the CHR program to use tail calls for the recursive constraints. The notion of tail calls in the context of CHR is explained later.

Even though similar tail call optimizations are possible in imperative host languages [46], in practice, most compilers for imperative languages do not perform them, or only in certain situations. The GCC C compiler [24], for instance, only optimizes tail calls in specific cases [8]. Most implementations of the Java Virtual Machine [41], including Sun's reference implementation HotSpot [68], do not perform (recursive) tail call optimizations at all[4]. Indeed, in practice we have observed that our naive compilation schema to Java overflows the execution stack very quickly. For C the situation is only slightly better.

Since improving the optimizations in the host language compilers is seldom an option, we designed novel compilation schemes that avoids execution stack overflows. Stack overflow can only occur when calling arbitrary host language code. Our new schema keeps data structures for the control flow of a CHR program on the heap. It might seem that the overflow is just shifted from the stack

[4] Supporting tail call optimization would interfere with Java's stack walking security mechanism (though this security folklore has recently been challenged in [12]).

to the heap. However, in the new schema we guarantee that these data structures remain constant size for CHR programs that are tail recursive. Moreover, in a language like Java, the heap is substantially larger than the call stack. So in any case, even for non-tail recursive CHR programs, the memory limits will be reached considerably later. This is also experimentally validated in Section 6.

Tail calls in CHR In CHR, a *tail call* occurs when the active constraint matches a removed occurrence, and the body ends with the addition of a CHR constraint. If the active constraint is not removed, the last body conjunct is not considered a tail call, as the search for partner constraints has to be resumed after the execution for the body, or more occurrences have to be tried for the previously active constraint.

Example 22. Recall the *add* rule of Example 10. For an active `pc(L)` constraint, the execution of this rule's body results in a tail call. The `leq(Y,Z)` constraint added by the body of the *transitivity* rule of Fig. 1, however, is not a tail call.

Using the optimized compilation schemes presented below, tail calls no longer consume space.

Trampoline Tail calls in CHR can be optimized such that they no longer consume stack space. A CHR constraint added by a tail call is no longer activated immediately by calling the corresponding occurrence procedures. Instead, a constraint suspension is returned that represents this constraint. Control then always returns to a loop that activates these suspensions as long tail calls occur. This technique is called *trampoline* [7,30].

So tail calls are replaced by a **return** of the newly created constraint. The '**return true**' statements introduced for the *Early Drop* and *Drop after Reactivation* optimizations (cf. Section 5.3) are replaced by '**return** `DROP`', the default '**return false**' statements by '**return nil**'. These are the only changes required to the occurrence procedures. All optimizations of Section 5.3 remain applicable.

Example 23. The RAM handler contains many tail calls. The compilation of the `pc(L)` occurrence of its *add* rule (see Example 10) using the trampoline compilation scheme is shown in Listing 10.

Next we modify the c`(X`$_1$`,...,X`$_n$`)` and `reactivate(ID)` procedures of Listing 2 to loop as long as the occurrence procedure returns a constraint suspension to activate. The looping is done by a separate procedure `trampoline`, called from both c and `reactivate`. The resulting compilation scheme is shown in Listing 11. The listing also shows the modified `activate` procedure. In the default case, **nil**, the next occurrence procedure is tried. Otherwise, the control returns to the trampoline (lines 11–15). The returned value is either a constraint suspension in case of a tail call, or the special DROP value. The latter case corresponds with a **Drop** transition of the current active constraint, so the trampoline exits. In the former case though, the constraint from the tail call is activated. By always returning to the trampoline this way, tail calls no longer consume stack space.

```
            ⋮
            kill(ID3)
            kill(ID4)
            mem(A,X+Y)
            return create(pc,[L+1])
          end
        ⋱
    end
    return nil
end
```

Listing 10. The compilation scheme for the RAM simulator rule occurrence, modifying Listing 9 for use in the trampoline scheme of Listing 11.

```
procedure c(X1,...,Xn)
    ID = create(c, [X1,...,Xn])
    trampoline(ID)
end

procedure reactivate(ID)
    trampoline(ID)
end

procedure trampoline(cont)
    do
        cont = activate(cont)
    while cont ≠ DROP
end

procedure activate(c(X1,...,Xn)#ID)
    ret = occurrence_c_1(ID,X1,...,Xn)
    if ret ≠ nil
        return ret
    end
    ⋮
    ret = occurrence_c_m(ID,X1,...,Xn)
    if ret ≠ nil
        return ret
    end
    store(ID)
    return DROP
end
```

Listing 11. Compilation scheme for an n-ary constraint `c` with m occurrences throughout the program, replacing Listing 2. A *trampoline* loop is added around the call to `activate`. The latter procedure is further modified to return either a constraint suspension, in case of a tail call, or the special DROP value otherwise.

Explicit stack This subsection presents a more general solution. Whilst trampoline-style compilation deals with tail calls only, the new compilation scheme deals with all instances of recursion. Instead of mapping the refined semantics' activation stack onto the host's implicit call stack, it maintains an explicit *continuation stack* on the heap. The elements on this stack are called *continuations*, and represent "the rest of the computation for an active constraint".

If a conjunction of multiple CHR constraints has to be executed, a *continuation* is pushed onto this stack, containing all information required to execute all but the first body conjunct. Next, the first conjunct is executed by returning to an outer control loop, similar to the trampoline scheme. After this conjunct has been handled completely, the continuation is popped from the stack, and the remainder of the body is executed in a similar, conjunct-by-conjunct fashion. If the remaining body is empty, and the active occurrence is alive, more partner constraints are searched, or the next occurrence is tried. Similar techniques are used to solve recursion involving built-in constraints or host language code.

We treat constraint suspensions as a special case of continuations. If *called* (see below), the corresponding constraint is simply activated. The following operations are introduced:

`push(⟨`ID`, occurrence number, body index, vars⟩)` Pushes a new *continuation* onto the continuation stack. This continuation contains the identifier of the active constraint, the number of the occurrence that caused the rule application, the index of the next body conjunct to be executed, and the variables required to execute the remainder of the body.

`push(`ID`)` Pushes a constraint suspension on the continuation stack.

`pop()` Removes the most recently pushed continuation from the continuation stack and returns it.

`call(`*`continuation`*`)` An extension of the `activate` operation of Listing 11. For constraint suspensions, `call` is equivalent to `activate`. Calling another constraint suspension resumes the handling of a suspended active constraint. This entails executing any remaining body conjuncts, resuming a search for partner constraints, or advancing to the next occurrence. The implementation of this operation is discussed below.

The compilation scheme is listed in Listing 12. Similar to the trampoline compilation scheme, recursion is solved by always returning to an outer control loop. The main differences with Listing 11 are the generalization of constraint suspensions to continuations, and the use of the continuation stack. As before, a constraint suspension for the newly told constraint is created (line 3), followed by a loop that repeatedly calls continuations (lines 4–9). Calling a continuation still returns the next continuation to be executed, which is DROP if the handling of the previously active constraint is finished. In the latter case, a next continuation is popped from the stack (lines 6–8).

To overcome recursion involving multiple constraint CHR handlers, or interleaving with host language code (see later), all cooperating CHR handlers of the same constraint system share the same continuation stack (cf. Section 4.2). In order to know when to return from the procedure of Listing 12, a special

```
1  procedure c(X_1,...,X_n)
2     push(SENTINEL)
3     cont = create(c, [X_1,...,X_n])
4     do
5        cont = call(cont)
6        if cont = DROP
7           cont = pop()
8        end
9     while cont ≠ SENTINEL
10 end
11
12 procedure reactivate(ID)
13    push(ID)
14 end
```

Listing 12. Compilation scheme for telling a CHR constraint using a continuation stack.

SENTINEL continuation is pushed on line 2. Popping a SENTINEL continuation means the **Drop** transition for the constraint initially told by the procedure was reached (line 6), and the procedure must return.

In the remainder of this section, we focus on the compilation scheme for the piecewise execution of the body. For a more complete discussion of the compilation scheme and the `call` operation, we refer to [70].

Recall the generic form of a CHR rule ρ introduced in Section 5.2:

$$\rho \,@\, \mathrm{c}_1^{[j_1]}(\mathrm{X}_{1,1}, \ldots, \mathrm{X}_{1,a_1}), \ldots, \mathrm{c}_{r-1}^{[j_{r-1}]}(\mathrm{X}_{r-1,1}, \ldots, \mathrm{X}_{r-1,a_{r-1}}) \setminus \\ \mathrm{c}_r^{[j_r]}(\mathrm{X}_{r,1}, \ldots, \mathrm{X}_{r,a_r}), \ldots, \mathrm{c}_h^{[j_h]}(\mathrm{X}_{h,1}, \ldots, \mathrm{X}_{h,a_h}) \Leftrightarrow \mathrm{g}_1, \ldots, \mathrm{g}_{n_g} \mid \mathrm{b}_1, \ldots, \mathrm{b}_{n_b}.$$

Suppose an applicable rule was found with the active constraint matching the $\mathrm{c}_i^{[j_i]}$ occurrence. Suppose the body conjuncts that still have to be executed are $\mathtt{b_k}, \ldots, \mathtt{b_{n_b}}$, with $k \leq n_b$. At the **Propagate** or **Simplify** transition itself, k will be equal to one, but when calling a continuation k can be larger than one. We distinguish three different cases:

(1) $\mathtt{b_k}$ *is a CHR constraint*
Let $\mathtt{b_k} = c(Y_1, \ldots, Y_a)$, then this case is simply implemented as:

```
    ⋮
    push(ID_i, j_i, k+1, vars⟨b_{k+1},...,b_{n_b}⟩)
    return create(c, [Y_1,...,Y_a])
.·.
```

The constraint suspension of the first conjunct is returned, after pushing a continuation on the stack. The expression $vars\langle b_{k+1}, \ldots, b_{n_b}\rangle$ returns all variables required in the remainder of the body. This way, after the returned constraint is activated and fully handled, the execution continues handling the currently

active constraint. First, the remaining body will be piecewise executed, using the compilation scheme presented in this section. If the body is completely executed, and the active constraint is still alive, more applicable rules are searched, starting at the occurrence that caused the previous rule application.

Example 24. In Listing 13 the full generated pseudocode for our running example is given. Only the code for the occurrence of the `pc` constraint in the *add* rule is given. The body of the occurrence first creates a continuation on the stack, and then returns the `mem` constraint that needs to be activated next. Eventually, after this constraint is completely handled, the continuation will be popped from the stack and executed. The code for this continuation, given in lines 45–47, simply returns the new `pc` constraint.

(2) $\texttt{b}_\texttt{k}$ *is a built-in constraint*

Calling a built-in constraint from a body may reactivate CHR constraints, which could cause recursive applications of the same rule. To avoid this, the `reactivate(ID)` procedure of Listing 12 is simply implemented as `push(ID)`. Reactivations are thus not performed immediately, but instead pushed onto the continuation stack. A built-in constraint $\texttt{b}_\texttt{k}$ in a body is then compiled as:

```
    ⋮
    push(ID_i, j_i, k + 1, vars⟨b_{k+1}, ..., b_{n_b}⟩)
    b_k
    return pop()
...
```

The continuation is pushed, the built-in constraint is executed, and the top of the stack is returned. If the built-in constraint triggered reactivations these are executed first. If not, the continuation itself is popped and executed. Notice that compared to the scheme that used the native call stack, this scheme reverses the order in which constraints are reactivated. This is allowed because the refined operational semantics does not determine this order.

If built-in constraints only rarely reactivate constraints, the above scheme is overly expensive. The creation, pushing and popping of the continuation can be avoided. One possible optimization uses the following two stack operations:

`stackSize()` Returns the number of continuations currently on the stack.

`replace(index, ID, occurrence number, body index, vars)`
: Similar to `push`, but adds the continuation on a given index rather than on top of the stack. The operation returns the continuation that was previously on the given index.

The stack indexes start from zero, so the index of the next continuation to be pushed is equal to `stackSize`. In other words, '`replace(stackSize(), ...)`' is equivalent to '`push(...)`'. The compilation scheme becomes:

```
    ⋮
    SS = stackSize()
```

```
1  procedure pc(L)
2     push(SENTINEL)
3     cont = create(pc, [L])
4     do
5        cont = call(cont)
6        if cont = DROP
7           cont = pop()
8        end
9     while cont ≠ SENTINEL
10 end
11
12 procedure call(pc(L)#ID)
13    ret = occurrence_pc_1(ID,L)
14    if ret ≠ nil
15       return ret
16    end
17    ⋮
18    ret = occurrence_pc_m(ID,L)
19    if ret ≠ nil
20       return ret
21    end
22    store(ID)
23    return DROP
24 end
25
26 procedure occurrence_pc_1(ID4,L)
27    prog(L1,$1,B1,A)#ID2 = lookup_single(prog,{L=L1})
28    if ID2 ≠ nil
29       if $1 = ADD
30          mem(A1,X)#ID3 = lookup_single(mem,{A=A1})
31          if ID3 ≠ nil
32             mem(B,Y)#ID1 = lookup_single(mem,{B=B1})
33             if ID1 ≠ nil and ID1 ≠ ID3
34                kill(ID3)
35                kill(ID4)
36                push(⟨pc,1,2,[L]⟩)
37                return create(mem,[A,X+Y])
38             end
39          end
40       end
41    end
42    return false
43 end
44
45 procedure call(⟨pc,1,2,[L]⟩)
46    return create(pc,[L+1])
47 end
```

Listing 13. Full example of generated code for the `pc(L)` constraint in the RAM simulator. Included are the `pc(L)` procedure for adding a constraint to the store from host language, occurrence code for first occurrence, and a polymorphic `call` dispatcher for both `pc` suspensions and continuations of the first occurrence body.

```
⋮
b_k
if (stackSize() > SS)
   return replace(SS, ID_i, j_i, k + 1, vars⟨b_{k+1}, ..., b_{n_b}⟩)
⋮ /* remainder of the body, starting with b_{k+1} */
```

This way, a continuation is only created and pushed, if a built-in constraint causes reactivations. By remembering the old stack size, this continuation is inserted exactly on the same location as before. The result of the call to `replace` will be a reactivation continuation. If no reactivations are pushed though, the control simply continues with the remainder of the body, or with the search for partner constraints.

(3) `b_k` *is a host language statement*

Recursion where CHR code is interleaved with host language code is more difficult to eliminate. Firstly, host language code may not only reactivate CHR constraints, it can also add new CHR constraints. The scheme used for built-in constraints therefore cannot be used, as reversing the activation order of CHR constraints told from host language code is not allowed by the refined operational semantics. Secondly, CHR handlers and built-in constraint solvers are *incremental*: executing a built-in or CHR constraint has to immediately perform all required changes, before returning control. By default, CHR constraint solvers should remain incremental, as host language code may rely on this property. This is also why the SENTINEL continuation is pushed on line 2 of Listing 12: this way multiple activations that have to return control after their activation can be handled using the same stack.

The implicit call stack can still overflow if a CHR handler and host language code recursively call each other. Arguably, this behavior is acceptable. One possibility to safeguard against these stack overflows though is to abandon incrementality. A queue can then be used to collect all constraints told whilst executing a host language statement in a body. Once the control returns to the CHR handler, the enqueued continuations are pushed, *in reverse order*, on the continuation stack. We refer to [70] for more details.

Optimizations If the active constraint is still alive after the execution of a body, remaining partner constraints have to be searched. The naive explicit stack scheme outlined in the previous subsection simply restarts all iterations, relying on the propagation history to avoid duplicate rule applications. This results in many redundant lookups, iterations, and history checks. The optimized scheme includes the constraint iterators into the continuations of the explicit stack, and uses them to efficiently resume the search for partner constraints.

Explicitly maintaining a stack unavoidably entails constant time overheads when compared to the traditional, call-based compilation scheme. The host environment's call stack is able to use more specialized low level mechanisms. This is particularly the case for high-level host languages such as Java. Possible optimizations to reduce these overheads include:

– Pushing a continuation can sometimes be avoided,for instance in the case of a tail call. Also, if there are no more partner constraints to be searched due to set semantics, the pushing of a continuation at the end of the body can be improved. These specializations can be done either statically, or by simple runtime checks.
– If static analysis shows activating a constraint does not result in recursion, the constraint can simply be activated using the compilation scheme of Sections 5.2–5.3.
– Continuations can sometimes be reused.
– The *Drop after Reactivation* optimization can be generalized to not only drop if a constraint is reactivated during an activation, but also if it is reactivated during an earlier reactivation. As constraint reactivations are not executed immediately, but instead scheduled on the continuation stack. This can again lead to the same constraint occurring multiple times on the continuation stack. In the scheme outlined above, these redundant reactivations can easily be avoided[5].

For a more detailed description of these and other optimizations, we refer to [70].

Example 25. In Listing 13, creating the continuation on line 36 is not necessary. As the remaining body is a tail call, simply pushing the constraint suspension representing the `pc(L+1)` constraint suffices.

Furthermore, after application of the *Passive Occurrences* optimization (see Section 5.3), most occurrences of the *mem* constraint are passive in the RAM handler (Appendix A). Static analysis therefore easily derives that adding a *mem* never causes recursion. A *mem* constraint can therefore safely be activated using the host's call stack.

When combining these optimizations, Listing 13 essentially reduces to Listing 10. For the RAM handler the explicit call stack is thus never used.

Conclusion The implicit call stack of the host environment is replaced by an explicitly maintained stack on the host's heap. If the explicit stack is not used though, the compilation scheme of Listing 12 becomes equivalent to the trampoline scheme of Listing 11. CHR tail calls therefore do not consume space. Also, even if tail optimizations are not possible, the heap of imperative hosts such as Java or C is considerably larger than their stack. We refer to Section 6 for an experimental evaluation.

6 Evaluation

Using the compilation scheme given in Section 5, we implemented a CHR embedding for two imperative host languages, Java and C. These implementations are briefly discussed in Section 6.1, and their performance is evaluated in Section 6.2.

[5] This optimization is not specific to the compilation scheme with an explicit stack. The implementation for the scheme presented in Sections 5.2–5.3 however is less straightforward.

6.1 Implementations

In this section we briefly discuss the implementation of two imperative embeddings of CHR, namely the K.U.Leuven JCHR system [75] for Java and CCHR [82] for C. These implementations are available at respectively [71] and [81].

The most important language design issues taken for both systems are discussed in Section 4. Our implementations do not provide search. We are convinced though that search can be added effectively to our CHR systems as a mostly orthogonal component, as shown by related systems [39,79].

Both systems implement the compilation scheme presented in Section 5. The implemented optimizations are listed in Table 1. As a reference, the table also lists the optimizations implemented by the K.U.Leuven CHR system [51,53] for SWI-Prolog [77] (in the version of SWI used, the memory reuse optimizations of [61] were not implemented).

Optimization	**Prolog**	**JCHR**	**CCHR**
Loop-Invariant Code Motion	✓	✓	✓
Indexing	✓	✓	✓
Join Ordering	✓	✓	✓
Set Semantics		✓	
Early Drop	✓	✓	✓
Backjumping		✓	✓
Non-Robust Iterators	✓	✓	✓
Late Storage	✓	✓	±
Late Allocation	✓		±
Distributed Propagation History	✓	✓	✓
History Elimination	✓	✓	
Guard Simplification	✓		
Passive Occurrences	✓	✓	±
Selective Constraint Reactivation	✓	✓	✓
Delay Avoidance	✓	✓	
Memory Reuse			±
Generations	✓	✓	✓
Recursion optimization		✓	✓

Table 1. Summary of all listed optimizations and their implementations in K.U.Leuven CHR for SWI- and YAP Prolog, K.U.Leuven JCHR and CCHR (development versions of March 1, 2008). Optimizations that are implemented only partially or in an ad-hoc fashion are indicated with '±'.

As discussed in Section 5.3, the recursion optimizations are less relevant for a Prolog implementation, as the Prolog runtime performs tail call optimizations. Both JCHR and CCHR explicitly maintain an explicit continuation stack when necessary (see Section 5.3). In CCHR continuations are efficiently implemented using `goto` statements. In Java this is not possible. For a detailed discussion of the compilation scheme used by JCHR, we refer to [70].

6.2 Performance

To verify our implementation's competitiveness, we benchmarked the performance of some typical CHR programs. The following benchmarks were used[6]:

- Calculating $tak(500, 450, 405)$ with a tabling Takeuchi function evaluator.
- Using Dijkstra's algorithm to find the shortest path in a sparse graph with 16,384 nodes and 65,536 edges. A Fibonacci heap, also implemented in CHR, is used to obtain the optimal complexity (see [60] for a description of the Dijkstra and Fibonacci heap handlers).
- Solving a circular set of 100 less-or-equal-than constraints (see Fig. 1).
- Calculating 25,000 resp. 200,000 Fibonacci numbers using the RAM simulator (see Appendix A), with the addition replaced by a multiplication to avoid arithmetic operations on large numbers (when using multiplication all Fibonacci numbers are equal to one).

The results[7] can be found in Table 2. We compared our two systems with the K.U.Leuven CHR system implementation for SWI-Prolog, and its port to YAP Prolog [49], a more efficient Prolog system. The YAP implementation used an older version of K.U.Leuven CHR. Execution times for native implementations in C and Java were added for reference.

	Takeuchi	**Dijkstra**	**leq**	**RAM**	
				25k	**200k**
YAP	2,310 (100%)	44,000 (100%)	4,110 (100%)	1,760 (100%)	15,700 (100%)
SWI	3,930 (170%)	6,620 (15%)	17,800 (433%)	1,000 (57%)	*stack overflow*
CCHR	48 (2.1%)	1,170 (2.7%)	189 (4.5%)	416 (24%)	3,540 (23%)
JCHR	183 (7.9%)	704 (1.6%)	68 (1.7%)	157 (8.9%)	1,714 (11%)
C	10 (0.4%)	-	2 (.05%)	1.3 (.07%)	12.7 (.08%)
Java	11 (0.5%)	-	2 (.05%)	2 (.11%)	16 (.10%)

Table 2. Benchmark comparing performance in some typical CHR programs in several systems. The average CPU runtime in milliseconds is given and, between parentheses, the relative performance with YAP Prolog as the reference system.

The imperative systems are significantly faster than both Prolog systems, up to one or two orders of magnitude, depending on the benchmark. This is partly due to the fact that the generated Java and C code is (just-in-time) compiled, whereas the Prolog code is interpreted. In SWI the RAM benchmark consumes linear stack space as the SWI runtime does not perform the necessary tail call optimizations. The RAM benchmark for 200k Fibonacci numbers therefore results in a stack overflow.

[6] Benchmarks available at `http://www.cs.kuleuven.be/~petervw/bench/lnai2008/`

[7] The benchmarks were performed on a Intel® Core™2 Duo 6400 system with 2 GiB of RAM. SWI-Prolog 5.6.50 and YAP 5.1.2 were used. All C programs were compiled with GCC 4.1.3 [24]. K.U.Leuven JCHR 1.6.0 was used; the generated Java code was compiled with Sun's JDK 1.6.0 and executed with HotSpot JRE 1.6.0.

The native C and Java implementations remain two orders of magnitude faster than their CHR counterparts. The main reason is that these programs use specialized, low-level data structures, or exploit domain knowledge difficult to derive from the CHR program. The Dijkstra algorithm was not implemented natively.

To show the necessity for the recursion optimizations of Section 5.4 in Java and C, we also benchmarked the limits on recursion. A simple program was tested that recursively adds a single constraint. If this happened using a tail call, JCHR runs out of stack space after 3,200 steps when using the unoptimized compilation scheme (i.e., without recursions optimizations). For CCHR, the GCC compiler was able to perform tail call optimization. Both YAP and SWI performed tail call optimization as well. For these systems, the test therefore ran in constant stack space, without limits. Using the optimized compilation scheme of Section 5.4, the same applies of course for JCHR (and CCHR).

If the recursive call was not a tail call, the different systems showed the following limits. Both SWI's and Java's native call stack have static size. In SWI, the test resulted in a stack overflow after 3.3 million recursive calls, in JCHR, without recurions optimization, already after 3,200 calls (the same as when a tail call was used). These numbers clearly show the necessity for the recursion optimizations when compiling to Java. If using an explicit call stack, JCHR is only limited by available heap memory, which is substantially larger than the stack. Using standard heap size, more than 1.8 million calls were possible. As the Java system used can be configured to use larger heap sizes, JCHR became essentially only limited by available (virtual) memory. The size of Java's call stack could not be configured.

The results for C were similar to those for Java: when using the unoptimized scheme the C call stack overflowed after around half a million recursive calls, whereas with the explicit stack optimization, CCHR permits over 40 million recursive calls. YAP Prolog's call stack grows dynamically, so YAP is also only limited by available (virtual) memory.

7 Related work

Even though (C)LP remains the most common CHR host language paradigm (see e.g. [17,36,34,53,51]), an increasing number of other CHR implementations have appeared. In this section we discuss several CHR embeddings in functional and imperative host languages (Sections 7.1 and 7.2 respectively), focussing on how they deal with the issues raised in Section 3.

Of course, countless other declarative paradigms have been integrated and compiled to imperative host languages. We only consider *production rules* (Section 7.3), as this formalism is most closely related to CHR.

7.1 CHR in Functional Languages

When embedding CHR in functional languages, many of the same challenges are met. Typically, functional languages are statically typed, and do not provide

search or built-in constraints. Structural matching on compound data on the other hand is mostly readily available.

HCHR provides a type-safe embedding of CHR in Haskell, leveraging the the built-in Haskell type checker to type check HCHR programs [11]. HCHR constraints only range over typed logical variables and terms, encoded as a polymorphic Haskell data type. Unification and matching functions are generated automatically for each type (this is similar to the approach taken by CCHR, cf. Section 4.3). Haskell data structures therefore have to be encoded as terms when used in a HCHR constraint, and reconstructed again when retrieving answers. No Haskell functions can be called from HCHR rule bodies, probably due to this data type mismatch.

The Chameleon system [66] is a Haskell-style language that incorporates CHR. It has been applied successfully to experiment with advanced type system extensions [65]. Chameleon's back-end CHR solver is HaskellCHR [16]. To allow Prolog-style terms with variables, this system includes a WAM implementation for Haskell, written in C. It is the only Haskell CHR system to provide chronological backtracking. HaskellCHR is not intended to be used stand-alone, but simply as a back-end to Chameleon.

With the advent of software transactional memories (STM) in Haskell, two systems with parallel execution strategies have recently been developed: Concurrent CHR [40] and STMCHR[64]. These systems are currently the only known CHR implementations that exploit the inherent parallelism in CHR programs.

Even though both HCHR and Chameleon provide syntactic preprocessing, both Haskell implementations are fairly naive interpreters. Their performance cannot compete with the optimizing CHR compilers for logic and imperative programming host languages. Similarly, the STM implementations are still early prototypes, whose performance is not yet competitive with sequential state-of-the-art implementations (unless of course when multiple processors are used for highly parallelizable CHR programs).

7.2 CHR in Imperative Languages

Aside from the systems discussed in this article, there exist at least three other CHR systems in Java. The oldest is the Java Constraint (JaCK) [2,4]. The main issue with JaCK is probably its lacking performance (see e.g. [75]). The JaCK framework consists of three major components:

JCHR A CHR dialect intended to be very similar to Java, in order to provide an intuitive programming language [50]. The semantics of the language are unspecified, and are known to deviate from other CHR implementations. CHR constraints only range over typed logical variables. All Java objects thus have to be wrapped in logical variables. Only static Java methods can be called from a rule's body.

VisualCHR An interactive tool visualizing the execution of JCHR [5]. It can be used to debug and to improve the performance of constraint solvers.

JASE , the Java Abstract Search Engine, allows for a flexible specification of tree-based search strategies [39]. JCHR bodies do not contain disjunctions, as in (C)LP implementations of CHR$^\vee$. Instead, JASE is added to JaCK as an orthogonal component. The JASE library provides a number of utility classes that help the user to implement a search algorithm in Java. A typical algorithm consists of the following two operations, executed in a loop: first, a JCHR handler is run until it reaches a fix-point, after which a new choice is made. If an inconsistency is found, chronological backtracking is used to return to the previous choice point. JASE aids in maintaining the search tree, and can be configured to use either trailing or copying.

The CHORD system (Constraint Handling Object-oriented Rules with Disjunctive bodies) [76], developed as part of the ORCAS project [47], is a Java implementation of CHR$^\vee$ [42]. Its implementation seems to build on that of JaCK, but adds the possibility to include disjunction in rule bodies.

A last CHR system for Java is DJCHR (Dynamic JCHR) [78], which implements an extension of CHR known as adaptive CHR [80]. Constraint solving in a dynamic environment often requires immediate adaptation of solutions when constraints are added or removed. By nature, CHR solvers already support efficient adaptation on constraint addition. Adaptive CHR is an extension of CHR capable of adapting CHR derivations after constraint deletions as well [80].

Constraints in DJCHR range only over Herbrand terms. Integration of the host language in the CHR rules is not supported. The system seems mainly created to experiment with the incremental adaptation algorithm of [80]. Like JaCK, DJCHR was later extended to support a wide range of search strategies [79]. Search is again implemented orthogonally to the actual CHR handlers. Interestingly, [79] clearly shows that the use of advanced search strategies can be more efficient than a low-level, host language implementation of chronological backtracking (as in Prolog).

7.3 Production Rules

Production rules, or *business rules* as they are often called, are a forward chaining rule-based programming language extension, very similar to CHR. Most of the many commercial and open-source implementations of this paradigm are based on the classical RETE algorithm [23]. This algorithm eagerly computes and maintains all possible joins and applicable rules. As the first rule fired with a fact (the equivalent of a CHR constraint) often already removes this fact (see also Section 5.3), RETE can be very costly. A lazy algorithm that fires applicable rules as it finds them, is usually more efficient in both time and memory. Even though this fact has been recognized in the production rule literature [43], the RETE algorithm remains the most widely used implementation technique. We believe that the compilation scheme developed for CHR, as presented in this article, is the first rigorously studied lazy execution mechanism for forward chaining rules. It would be interesting to compare the performance of CHR with that of state-of-the-art, RETE based production rule engines.

Modern production rule systems such as Jess [25] and Drools [37] allow arbitrary host language code to be called. To solve the modification problem, a technique called *shadow facts* is commonly used. Consistency of the RETE network is maintained by keeping a clone of any modifiable object referred to by facts in an internal data structure. The user is responsible for notifying the rule engine of any changes to these objects. This solution however does not work for arbitrary objects (e.g. only for Java Bean objects [69]), and is fairly inefficient.

8 Conclusions and Future Work

In this work we presented our approach to solve the impedance mismatch between CHR and imperative languages. We outlined the different language design issues faced when embedding CHR into an imperative host language. In our approach, we advocated a tight and natural integration of both paradigms. We illustrated with two case studies, the K.U.Leuven JCHR system and CCHR, and showed that our approach leads to a programming language extension intuitive and useful to adepts of both CHR and the intuitive host language.

We ported the standard CHR compilation scheme to an imperative setting, and showed how the many existing optimizations can be incorporated. The result is a first comprehensive survey of the vast, recent literature on optimized CHR compilation. Many of the presented compilation and optimization techniques are applicable for any implementation of CHR, or any similar rule-based language.

More specific to imperative target languages, we showed that the standard call-based compilation scheme of CHR results in call stack overflows when used to compile to imperative host languages. We proposed a novel, optimized compilation scheme using which CHR programs written using tail calls are guaranteed to execute in constant space. Where tail call optimization is not possible, an explicitly maintained stack is used instead of the host's call stack. By maintaining the stack on the heap, memory limits are reached considerably later for all recursive CHR programs.

We created efficient, state-of-the-art implementations for Java and C, and showed that they outperform other CHR systems up to several orders of magnitude. We also showed the effectiveness of our recursion optimizations.

8.1 Future Work

Certain issues raised in Section 3 are not yet adequately solved by current, imperative CHR systems. The modification problem is only solved effectively for built-in constraints. Similarly, the combination of arbitrary host language code with search requires more investigation. The integration of search in an efficient compilation scheme is also an interesting topic for future research.

The current compilation scheme considers each occurrence separately. However, we believe that more efficient code can be generated with a more global compilation scheme. For instance, the entire RAM handler of Appendix A could be compiled to a single switch statement in a loop. Rather than linearly going

through all possible operations for each program counter, the applicable rule would be found in constant time using a switch. Sharing partial joins for overlapping rules among different occurrences is another example.

Over the past decade there has been an increase in the number of CHR systems. The support for advanced software development tools, such as debuggers, refactoring tools, and automated analysis tools, lags behind, and remains an important challenge, not only for the systems embedding CHR in imperative hosts, but for the entire CHR community.

References

1. The Constraint Handling Rules (CHR) programming language homepage, 2008. `http://www.cs.kuleuven.be/~dtai/projects/CHR/`.
2. S. Abdennadher. *Rule-based Constraint Programming: Theory and Practice*. Habilitationsschrift, Institute of Computer Science, LMU, Munich, Germany, July 2001.
3. S. Abdennadher, T. Frühwirth, and C. Holzbaur, editors. *Special Issue on Constraint Handling Rules*, volume 5(4–5) of *Theory and Practice of Logic Programming*. Cambridge University Press, July 2005.
4. S. Abdennadher, E. Krämer, M. Saft, and M. Schmauß. JACK: A Java Constraint Kit. In M. Hanus, editor, *WFLP '01: Proc. 10th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, volume 64 of *ENTCS*, pages 1–17, Kiel, Germany, Nov. 2002. Elsevier. See also `http://pms.ifi.lmu.de/software/jack/`.
5. S. Abdennadher and M. Saft. A visualization tool for Constraint Handling Rules. In A. Kusalik, editor, *WLPE '01*, Paphos, Cyprus, Dec. 2001.
6. S. Abdennadher and H. Schütz. $\text{CHR}^{\vee}$, a flexible query language. In T. Andreasen, H. Christiansen, and H. Larsen, editors, *FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems*, volume 1495 of *LNAI*, pages 1–14, Roskilde, Denmark, May 1998. Springer.
7. H. G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *SIGPLAN Notices*, 30(9):17–20, 1995.
8. A. Bauer. Compilation of functional programming languages using GCC—Tail calls. Master's thesis, Institut für Informatik, Technische Univ. München, 2003.
9. J. Bouaud and R. Voyer. Behavioral match: Embedding production systems and objects. In Pachet [45].
10. G. Bracha. *Generics in the Java Programming Language*, July 2004. Tutorial.
11. W.-N. Chin, M. Sulzmann, and M. Wang. A type-safe embedding of Constraint Handling Rules into Haskell. Honors thesis, School of Computing, National University of Singapore, 2003.
12. J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Trans. on Prog. Languages and Systems (TOPLAS)*, 26(6):1029–1052, 2004.
13. C. S. da Figueira Filho and G. L. Ramalho. JEOPS - the Java Embedded Object Production System. In *Advances in Artificial Intelligence — IBERAMIA-SBIA 2000: Proc. Intl. Joint Conf. 7th Ibero-American Conference on AI – 15th Brazilian Symposium on AI*, volume 1952 of *LNCS*, Atibaia, SP, Brazil, 2000. Springer.
14. L. De Koninck and J. Sneyers. Join ordering for Constraint Handling Rules. In Djelloul et al. [15], pages 107–121.

15. K. Djelloul, G. J. Duck, and M. Sulzmann, editors. *CHR '07: Proc. 4th Workshop on Constraint Handling Rules*, Porto, Portugal, Sept. 2007.
16. G. J. Duck. HaskellCHR. `http://www.cs.mu.oz.au/~gjd/haskellchr/`, 2004.
17. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Australia, Dec. 2005.
18. G. J. Duck and T. Schrijvers. Accurate functional dependency analysis for Constraint Handling Rules. In Schrijvers and Frühwirth [55], pages 109–124.
19. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP '03*, pages 79–90, Uppsala, Sweden, 2003. ACM Press.
20. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04*, volume 3132 of *LNCS*, pages 90–104, Saint-Malo, France, Sept. 2004. Springer.
21. F. Fages, C. M. de Oliveira Rodrigues, and T. Martinez. Modular CHR with *ask* and *tell*. In Schrijvers et al. [56], pages 95–110.
22. M. Fink, H. Tompits, and S. Woltran, editors. *WLP '06: Proc. 20th Workshop on Logic Programming*, T.U.Wien, Austria, INFSYS Research report 1843-06-02, Vienna, Austria, Feb. 2006.
23. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
24. Free Software Foundation. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`, 2008.
25. E. Friedman-Hill et al. Jess, the rule engine for the Java platform. `http://www.jessrules.com/`, 2008.
26. T. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
27. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, June 2009. To appear.
28. T. Frühwirth and M. Meister, editors. *CHR '04: 1st Workshop on Constraint Handling Rules: Selected Contributions*, Ulm, Germany, May 2004.
29. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
30. S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Intl. Conf. on Functional Programming*, pages 18–27, 1999.
31. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Prentice Hall, third edition, 2005.
32. C. Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In *Proc. 4th Intl. Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. Springer, 1992.
33. C. Holzbaur and T. Frühwirth. Compiling Constraint Handling Rules into Prolog with attributed variables. In G. Nadathur, editor, *PPDP '99*, volume 1702 of *LNCS*, pages 117–133, Paris, France, 1999. Springer.
34. C. Holzbaur and T. Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. In Holzbaur and Frühwirth [35], pages 369–388.
35. C. Holzbaur and T. Frühwirth, editors. *Special Issue on Constraint Handling Rules*, volume 14(4) of *Journal of Applied Artificial Intelligence*. Taylor & Francis, Apr. 2000.
36. C. Holzbaur, M. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. In Abdennadher et al. [3], pages 503–531.

37. JBoss. Drools. `http://labs.jboss.com/drools/`, 2008.
38. B. W. Kernighan, D. Ritchie, and D. M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall PTR, March 1988.
39. E. Krämer. A generic search engine for a Java Constraint Kit. Diplomarbeit, Institute of Computer Science, LMU, Munich, Germany, Jan. 2001.
40. E. S. Lam and M. Sulzmann. A concurrent Constraint Handling Rules semantics and its implementation with software transactional memory. In *DAMP '07: Proc. ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, Nice, France, Jan. 2007. ACM Press.
41. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Prentice Hall, 2 edition, 1999.
42. L. Menezes, J. Vitorino, and M. Aurelio. A high performance CHR$^\vee$ execution engine. In Schrijvers and Frühwirth [55], pages 35–45.
43. D. P. Miranker, D. A. Brant, B. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *Proc. 8th Intl. Conf. on Artificial Intelligence*, pages 685–692, 1990.
44. F. Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4):19–24, 1995.
45. F. Pachet, editor. *EOOPS'94: Proc. OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems*, Portland, Oregon, USA, Oct. 2004.
46. M. Probst. Proper tail recursion in C. Diplomarbeit, Institute of Computer Languages, Vienna University of Technology, 2001.
47. J. Robin and J. Vitorino. ORCAS: Towards a CHR-based model-driven framework of reusable reasoning components. In Fink et al. [22], pages 192–199.
48. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.
49. V. Santos Costa et al. YAP Prolog. `http://www.ncc.up.pt/yap/`.
50. M. Schmauß. An implementation of CHR in Java. Diplomarbeit, Institute of Computer Science, LMU, Munich, Germany, Nov. 1999.
51. T. Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.
52. T. Schrijvers and B. Demoen. Antimonotony-based delay avoidance for CHR. Technical Report CW 385, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium, July 2004.
53. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In Frühwirth and Meister [28], pages 8–12.
54. T. Schrijvers, B. Demoen, G. J. Duck, P. J. Stuckey, and T. Frühwirth. Automatic implication checking for CHR constraints. In *RULE '05: 6th Intl. Workshop on Rule-Based Programming*, volume 147(1) of *ENTCS*, pages 93–111, Nara, Japan, Jan. 2006. Elsevier.
55. T. Schrijvers and T. Frühwirth, editors. *CHR '05: Proc. 2nd Workshop on Constraint Handling Rules*, Sitges, Spain, 2005. K.U.Leuven, Dept. Comp. Sc., Technical report CW 421.
56. T. Schrijvers, F. Raiser, and T. Frühwirth, editors. *CHR '08: Proc. 5th Workshop on Constraint Handling Rules*, Hagenberg, Austria, July 2008. RISC Report Series 08-10, University of Linz, Austria.
57. T. Schrijvers, P. J. Stuckey, and G. J. Duck. Abstract interpretation for Constraint Handling Rules. In P. Barahona and A. Felty, editors, *PPDP '05*, pages 218–229, Lisbon, Portugal, July 2005. ACM Press.
58. C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. Under consideration for *ACM Transactions on Programming Languages and Systems*, 2008.

59. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. In Schrijvers and Frühwirth [55], pages 3–17.
60. J. Sneyers, T. Schrijvers, and B. Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In Fink et al. [22], pages 182–191.
61. J. Sneyers, T. Schrijvers, and B. Demoen. Memory reuse for CHR. In S. Etalle and M. Truszczynski, editors, *ICLP '06*, volume 4079 of *LNCS*, pages 72–86, Seattle, Washington, Aug. 2006. Springer.
62. J. Sneyers, T. Schrijvers, and B. Demoen. Guard reasoning in the refined operational semantics of CHR. volume 5388 of *LNAI*, pages 213–244. Springer, Dec. 2008.
63. J. Sneyers, P. Van Weert, T. Schrijvers, and L. De Koninck. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Submitted to *Journal of Theory and Practice of Logic Programming*, 2009.
64. M. Stahl. STMCHR. Available at the CHR Homepage [1], 2007.
65. P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM TOPLAS*, 27(6):1216–1269, 2005.
66. P. J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon system. In Frühwirth and Meister [28], pages 13–32.
67. Sun Microsystems, Inc. The Collections framework: API's and developer guides. `http://java.sun.com/javase/6/docs/technotes/guides/collections/`, 2008.
68. Sun Microsystems, Inc. Java SE HotSpot at a glance. `http://java.sun.com/javase/technologies/hotspot/`, 2008.
69. Sun Microsystems, Inc. JavaBeans. `http://java.sun.com/products/javabeans/`, 2008.
70. P. Van Weert. Compiling Constraint Handling Rules to Java: A reconstruction. Technical Report CW 521, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium, Aug. 2008.
71. P. Van Weert. The K.U.Leuven JCHR system. `http://www.cs.kuleuven.be/~petervw/JCHR/`, 2008.
72. P. Van Weert. *K.U.Leuven JCHR User's Manual*, 2008. Available at [71].
73. P. Van Weert. Optimization of CHR propagation rules. In M. García de la Banda and E. Pontelli, editors, *ICLP '08*, volume 5366 of *LNCS*, pages 485–500, Udine, Italy, Dec. 2008. Springer.
74. P. Van Weert. A tale of histories. In Schrijvers et al. [56], pages 79–94.
75. P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Schrijvers and Frühwirth [55], pages 47–62.
76. J. Vitorino and M. Aurelio. Chord. `http://chord.sourceforge.net/`, 2005.
77. J. Wielemaker. An overview of the swi-prolog programming environment. In *Proc. 13th Intl. Workshop on Logic Programming Environments*, Mumbai, India, 2003. System's home page at `http://www.swi-prolog.org/`.
78. A. Wolf. Adaptive constraint handling with CHR in Java. In T. Walsh, editor, *CP '01*, volume 2239 of *LNCS*, pages 256–270, Paphos, Cyprus, 2001. Springer.
79. A. Wolf. Intelligent search strategies based on adaptive Constraint Handling Rules. In Abdennadher et al. [3], pages 567–594.
80. A. Wolf, T. Gruenhagen, and U. Geske. On incremental adaptation of CHR derivations. In Holzbaur and Frühwirth [35], pages 389–416.
81. P. Wuille. CCHR: The fastest CHR implementation, in C. `http://www.cs.kuleuven.be/~pieterw/CCHR/`, 2008.
82. P. Wuille, T. Schrijvers, and B. Demoen. CCHR: the fastest CHR implementation, in C. In Djelloul et al. [15], pages 123–137.

A A RAM Simulator Written in CHR

Fig. 9 contains a CHR handler that implements a simulator for a standard RAM machine. The memory of the simulated RAM machine is represented as `mem` constraints, the instructions of the program it is executing as `prog` constraints. The current program counter is maintained as a `pc` constraint. Different CHR rule declares what has to be done for each instruction type. Example 10 in Section 5.2 discusses the rule for an ADD instruction in more detail.

Four extra rules are added to the original program, as it first appeared in [59]. The first three rules ensure that illegal combinations of constraints cannot occur; the last rule safeguards against invalid program counters. These four extra rules allow static program analysis to defer certain program properties essential for an efficient compilation of the program, as shown in Section 5.3.

```
// enforce functional dependencies:
mem(A,_), mem(A,_) <=> fail.
prog(L,_,_,_), prog(L,_,_,_) <=> fail.
pc(_), pc(_) <=> fail.

prog(L,ADD,B,A), mem(B,Y) \ mem(A,X), pc(L) <=> mem(A,X+Y), pc(L+1).
prog(L,SUB,B,A), mem(B,Y) \ mem(A,X), pc(L) <=> mem(A,X-Y), pc(L+1).
prog(L,MULT,B,A), mem(B,Y) \ mem(A,X), pc(L) <=> mem(A,X*Y), pc(L+1).
prog(L,DIV,B,A), mem(B,Y) \ mem(A,X), pc(L) <=> mem(A,X/Y), pc(L+1).

prog(L,MOVE,B,A), mem(B,X) \ mem(A,_), pc(L) <=> mem(A,X), pc(L+1).
prog(L,I_MOV,B,A), mem(B,C), mem(C,X) \ mem(A,_), pc(L) <=> mem(A,X), pc(L+1).
prog(L,MOV_I,B,A), mem(B,X), mem(A,C) \ mem(C,_), pc(L) <=> mem(C,X), pc(L+1).

prog(L,CONST,B,A) \ mem(A,_), pc(L) <=> mem(A,B), pc(L+1).
prog(L,INIT,A,_), mem(A,B) \ pc(L) <=> mem(B,0), pc(L+1).

prog(L,JUMP,_,A) \ pc(L) <=> pc(A).
prog(L,CJMP,R,A), mem(R,X) \ pc(L) <=> X == 0 | pc(A).
prog(L,CJMP,R,_), mem(R,X) \ pc(L) <=> X != 0 | pc(L+1).

prog(L,HALT,_,_) \ pc(L) <=> true.

// Safeguard against invalid program counter:
pc(_) <=> fail.
```

Fig. 9. A RAM machine simulator written in CHR.