Efficient Lazy Evaluation of Rule-Based Programs

Peter Van Weert

Abstract—Thirty years after Forgy's seminal dissertation, Rete remains the de facto standard matching algorithm. Despite promising research results, alternative algorithms such as TREAT and LEAPS have had little impact on modern production rule engines. Constraint Handling Rules (CHR) is a high-level, declarative programming language, similar to production rules. In recent years, CHR has increasingly been used in a wide range of general-purpose applications. State-of-the-art CHR systems use LEAPS-like lazy matching, and implement a large body of novel program analyses and optimization techniques to further improve performance. While obviously related, CHR and production rules research have mostly evolved independently from each other. With this article, we aim to foster cross-fertilization of implementation techniques. We provide a lucid, comprehensive overview of CHR's rule evaluation methodology, and survey recent contributions to the field of lazy matching. Our empirical evaluation confirms Rete-based engines would surely benefit from incorporating similar techniques and optimizations.

Index Terms—optimizing compilation, lazy evaluation, matching algorithms, production rules, Constraint Handling Rules

1 INTRODUCTION

With the advent of modern-day business rule management systems (BRMS), adoption of rule-based technologies has increased significantly. Rules are an established technology in finance, banking and insurance, and companies across the board are starting to use rules to maximize business agility. The underlying technology, production rules, however, is far from new: early systems include OPS5 [1] and CLIPS [2], [3]. Modern BRMSs though have made tremendous progress in terms of flexibility, user-friendliness, and ease-of-use.

Implementation techniques, on the other hand, have not evolved in pace. Thirty years after its conception, the predominant basis of rule engines today remains the Rete algorithm [4], [5]. Despite strong empirical evidence of superior matching algorithms [6], [7], [8], [9], [10], [11], nearly all current rule engines still use Rete-based pattern matching. Given the increasing reliance of companies on rule-based technologies, this discrepancy surely warrants further research attention.

A paradigm closely related to production rules is Constraint Handling Rules (CHR) [12]. Today, CHR is the subject of an active academic research field. We refer to [13] for a comprehensive survey. Driven by applications in constraint solving, computational linguistics, and multi-agent systems, considerable research has been devoted to the efficient compilation and execution of CHR programs. CHR’s standard evaluation strategy, [14], is similar to the LEAPS (Lazy Evaluation Algorithm for Production Systems) matching algorithm [8], [15]. Recently, CHR research has produced a large body of novel static program analyses and optimization techniques to further improve its performance [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. These efforts resulted in several very efficient CHR systems, embedded in Prolog [14], [27], HAL [18], Java [28], and C [29].

Over the past two decades, research and practice in CHR and production rules have progressed independently, with only very little cross-fertilization between the two fields. With this work, we aim to initiate the interchange of ideas between the two communities, in particular with respect to implementation techniques:

• For readers with a background in production rules, we provide a comprehensive compendium of compilation principles and optimization techniques developed for CHR. We provide many references to relevant CHR research literature, making this specialized literature more accessible to non-experts.
• We put CHR implementation research into a broader perspective by comparing evaluation techniques and performance with those of state-of-the-art mainstream rule engines. We also show how to extend CHR’s optimizing compilation scheme to expressive language features such as negation [30] and priorities [31], without sacrificing on efficiency.

The remainder of this article is organized as follows. In Section 2, we define an illustrative, simplified rule language. Next, in Sections 3–4 we provide a clear, thorough description of our evaluation methodology. Throughout, we employ a tutorial-style presentation, using high-level pseudocode and many clarifying examples to gradually introduce first the fundamental principles of our approach (Section 3), and then the more advanced analysis...
and optimization techniques (Section 4). We evaluate our approach empirically in Section 5, and compare it with related work in Section 6. Section 7 concludes, and briefly overviews ongoing research.

2 A CORE RULE LANGUAGE

As modern, real-life rule-based languages are relatively feature-rich, comprehensively covering all features is out of the scope of this article (see Section 6.1 for references to relevant literature). In this section, we therefore define a simplified core language, sufficiently rich to express most rule-based programs. It corresponds to the ground subset of CHR, extended with negation [30] and priorities [31]. The terminology, syntax and semantics used are a compromise between CHR and production rules.

2.1 Facts

Facts are similar to immutable objects in object-oriented languages, or rows in a relational database. Each fact is a runtime instance of a class. A class has a unique name, and declares a number of fields, each with a type and an optional field name. For this article, standard boolean, string, integer, and float types suffice. Once asserted, the values of a fact’s fields can no longer be changed.

Example 1: The following class declaration is adapted from the classical WALTZ production rule program [9]:

```
class edge(int p1, int p2, boolean joined, string label, boolean plotted)
```

The program implements Waltz’ seminal algorithm for interpreting line drawings of three-dimensional scenes [32]. An edge fact represents a labeled, directed edge between two points (a two-dimensional point is represented using a single integer number), with two additional boolean fields required by the algorithm.

Facts are asserted and retracted by rules. The multiset of all asserted facts is called the fact base. As in CHR, we choose to allow the fact base to contain multiple syntactically identical facts. This is unlike most production systems, where duplicate facts are removed implicitly. We assume the fact base is initialized using rules of form ‘main :: +init ⇒ ’, and that a single init fact is asserted at the beginning of a program’s execution.

2.2 Rules

A rule consists of a unique name, an optional priority (see Section 2.2.2), and a left- and right-hand side (LHS and RHS). The LHS is a conjunction that specifies the conditions under which a rule is applicable:

1) +c(…) or −c(…): a pattern c(…), consisting of a class name c, and a comma-separated list of expressions, one for each field, in the order the fields were declared by the class. The wildcard variable ‘_’ may be used for fields of no interest for the matching. The ‘+’ and ‘−’ modifiers are explained shortly.
2) ?(…): a guard is a conjunction of additional restrictions (boolean expressions) on matching facts.
3) ~(…): A negated conjunction of patterns and guards. For ease of presentation, we do not allow nesting of negated conjunctions.

For single-conjunct guards or negated conjunctions, the surrounding parentheses may be omitted. We use the modifier positive to denote syntactical elements that occur in the non-negated part of the LHS. At least one positive pattern must be specified.

The RHS of a rule consists of fact assertions of the form c(…). Our simplified core language, as CHR, has no explicit ‘retract’ or ‘modify’ constructs, as commonly found in production rule languages. Instead, when a rule fires, all facts matching LHS patterns preceded with ‘∼’ are retracted. In accordance with the traditional semantics of ‘modify’ [1], [3], modifications can be done by retracting a fact and asserting a modified one.

Example 2: The WALTZ program contains this rule:

```
make_L ::
+stage("detect junctions"),
-edge(Base, P1, false, L1, Plotted1),
-edge(Base, P2, false, L2, Plotted2), (P1 ≠ P2),
~(edge(Base, P3, ..., L), (P3 ≠ P1, P3 ≠ P2))
⇒
junction(Base, P1, P2, 0,"L"),
edge(Base, P1, true, L1, Plotted1),
edge(Base, P2, true, L2, Plotted2).
```

The first LHS conjunct specifies a stage fact has to be present, whose single field equals "detect junctions". The two remaining positive patterns declare the fact base must also contain two edge facts with equal p1 fields. Next, a guard states their p2 fields must be different. The last LHS conjunct is negated, and ensures the rule is only applicable if no third edge fact with the same p1 value Base is present. If the rule fires, the two matched edge facts are retracted, and modified versions are asserted in the RHS along with an additional junction fact.

Variable names are capitalized, and all other names start with a lower-case letter (cf. CHR). The scope of a variable appearing in a positive pattern is the entire rule. A variable introduced in a negated conjunction—i.e., a variable that does not occur in a positive pattern—can only be used in that negated conjunction. Guards cannot introduce new variables. Besides from variables, standard arithmetic and boolean expressions are supported.

2.2.1 Matching Semantics

Before considering a language’s implementation, the operational semantics of its programs must be specified. CHR’s semantics is typically defined as a formal state-transition system [30], [31], [33]. This facilitates reasoning about program execution on an abstract level, and allows for formal correctness proofs of compiler optimizations and analyses. For the purpose of this article, an informal discussion of the matching semantics suffices.

A rule instance consists of a rule’s name, and a sequence of asserted (and not yet retracted) facts, such that:

1) Each fact in the sequence matches with the corresponding positive pattern in the rule’s LHS (using a left-to-right order). During matching, all variables
are bound to the values of the corresponding fields. If a variable occurs more than once, this implies the corresponding fields must have equal values.

A subtle difference with most production systems is that a single fact may not appear more than once in an instance, that is: it is not allowed to match more than one pattern. Practice shows this is almost always the desired semantics (see Ex. 3 below), certainly when the fact base is a multiset.

1) All guard conditions are satisfied (evaluate to true) for the variable bindings implied by the matching.

2) For none of the the negated conjunctions a set of matching facts, different from those matched by the positive part, is present in the fact base.

Example 3: In WALTZ, the following invariant holds (see Section 4.1, Ex. 9): there is at most one edge fact with given \( p_1 \) and \( p_2 \) fields. Nevertheless, the guard \( (? (P_2 \neq P_1)) \) in the rule of Ex. 2 had to be introduced to the original production rule program [9] to prevent a single edge fact from matching multiple patterns. Such extraneous guards frequently have to be added to production rules. With our matching semantics this is no longer necessary. The two negated guards are similarly implied by our semantics, and could be omitted as well.

2.2.2 Priorities

Each rule has an optional priority (salience in production rule terms), which is a positive or negative integer value. The default priority is 0. A rule instance is applicable if no rule instance with higher priority exists. The choice between instances of the same priority is left unspecified.

Example 4: The rule in Ex. 2 has default priority 0. Declared explicitly, this becomes: make_L @ 0 :: ... We refer to [31] for a detailed discussion and many examples of priorities as a powerful control mechanism.

2.2.3 Reapplication prevention

Without negation, it is intuitively clear that no rule instance should fire more than once (a principle sometimes referred to as refraction [34]). The appropriate semantics for rules that contain negation, however, is somewhat less obvious. It is best introduced using an example:

\[ abc :: a, \sim (b, c) \Rightarrow b, c. \]

Say the rule fires for some fact \( a \), thus asserting \( b \) and \( c \) facts. Suppose that next some rule retracts \( b \), causing the \( abc \) rule to fire again with the same \( a \) fact. This is allowed, because that instance was not applicable prior to \( b \)'s retraction. But if now \( c \) is retracted as well, firing the same instance a third time is not allowed, as it already was applicable before the retraction. Intuitively, a rule instance may fire once each time it becomes applicable. For a more precise specification, we refer to [30].

3 Basic Compilation Methodology

CHR programs are typically compiled to host-language code, rather than being interpreted. This allows the generation of specialized, highly optimized code. We use imperative pseudo-code to illustrate the compilation scheme used by most current CHR systems. Earlier descriptions—such as [16] (for the constraint logic language HAL), [14], [19] (Prolog), and [35] (Java)—mostly detail concrete implementations, and do not cover negation, priorities, or recent advanced optimizations.

In Section 3.3, we introduce the basic compilation scheme, and extend it to deal with negation and priorities in Sections 3.4 and 3.5 (Section 3.5 is inspired by [31]). The scheme presented here is still fairly naive and inefficient. It is designed, however, to be clear, and obviously correct with respect to the semantics outlined in Section 2. In Section 4, we gradually transform it into a highly optimized, efficient compilation scheme.

Before introducing the compilation scheme, the principle data structures and operations it uses are described in Section 3.1, and some further terminology and notation is established in Section 3.2.

3.1 Principle Data Structures and Operations

3.1.1 The fact base

The central data structure is the fact base. Efficient storage and retrieval of facts is crucial for performance. The actual implementation of the fact base (and its indexes: see Section 4.2.4) is outside the scope of this article. Instead, we simply list its basic properties and operations.

When asserted, each fact is assigned a unique identifier, an increasing integer number that may also serve as a timestamp. In the pseudo-code used throughout this article, a fact is denoted as \( c(X) \# ID \). We often make the transition from an identifier \( ID \) to its corresponding fact implicitly (much like a pointer in imperative languages).

The basic fact base operations are the following:

- \( create(c, X) \) creates a new fact of given class \( c \) with arguments \( X \), and returns its identifier
- \( store(ID) \) adds the referenced fact (created earlier with the \( create \) operation) to the fact base
- \( kill(ID) \) removes the fact from the fact base data structures; subsequent calls to \( alive \) return false
- \( alive(ID) \) tests whether the corresponding fact is alive, that is: not yet killed by the \( kill \) operation
- \( lookup(c) \) Returns an iterator (see below) over all stored facts of class \( c \)

To iterate over (subsets of) the fact base, we use the well-known iterator abstraction. We require the iterators returned by \( lookup \) operations to have at least the following properties [36]:

- \( robust \): even if facts are asserted or retracted while a fact iteration is interrupted, iteration can be resumed from the point where it was suspended
- \( correct \): iterators only return live facts
- \( complete \): all facts stored at the moment of the iterator’s creation are returned at least once
- \( weakly duplicate-free \): an uninterrupted iteration does not contain duplicates; if the fact base is updated while iteration is suspended, facts returned prior to that suspension may be returned once more
Iterators are further preferred to be strongly duplicate-free, which requires that they return each fact at most once. Iterators offered by predefined data structures typically do not have all required properties. Iterators returned by the standard Java data structures [37], for instance, are not robust under modifications.

3.1.2 The history

The history is a set of rule instances, used to prevent unwanted reapplication (see Section 2.2.3). It supports the two obvious operations addToHistory(rule, ID) and inHistory(rule, ID), where ID denotes a sequence of fact identifiers. A third operation, cleanHistory(), removes all rule instances that are no longer applicable.

3.2 Rule Normal Form

For ease of presentation, all rules are assumed to be transformed into a normal form. An example to illustrate:

Example 5: The following rule occurs in the RAM simulator program, used in [38] to show the complexity-wise completeness of CHR:

```plaintext
add :: -pc(L), +prog(L,"add",B,A),
     -mem(A,X), +mem(B,Y) ⇒ mem(A,X+Y), pc(L+1).
```

This rule simulates the ADD instruction of a Random Access Machine. The RAM machine’s program is represented as prog facts; its current program counter L is maintained in a single pc fact. If L refers to an ADD instruction, the above rule may apply. It retrieves two cells of the RAM machine’s memory, and replaces one of them with a mem cell containing the sum of their values, before advancing to the next instruction.

The rule’s normal form is as follows:

```plaintext
add :: -pc(L), -mem(A,X),
     +prog(L1, I, B, A1), +mem(B1, Y),
     (?(L1 = L, I = "add", B1 = B, A1 = A)
      ⇒ mem(A, X+Y), pc(L+1)).
```

In this normal form, all guards are made explicit such that all patterns only contain mutually distinct variables. Such patterns are said to be linear. Moreover, conjuncts of the same kind are grouped together: first all retracted patterns, followed by the kept patterns, the guards, and finally the similarly linearized negated conjunctions.

Henceforth, we will refer to (normalized) LHS patterns as occurrences. The rule in Ex. 5 for instance, contains two positive occurrences of class mem, one retracted and one kept occurrence. It contains no negative occurrences (patterns occurring in a negated conjunction).

More formally, our presentation of the general compilation scheme uses the following rule normal form:

```plaintext
@ p ::
   -c1(X1),..., -cp(Xp),
   +cp+1(Xp+1),..., +cn(Xn),
   (?(g1,...,gm), N1,...,Nm)
   ⇒ c1(Y1),..., cn(Yn).
```

with ~N1 = ~c1(X1,1),...,c1,n1(X1,n1), ?(g1,1,...,gm,1))

All LHS patterns are linearized. In other words: all X's denote sequences of mutually distinct variables.

---

**Procedure** occurrence(c_i,j(ID, X_i))

```plaintext
foreach c1(X1)#ID in lookup(c1)
  ...
   foreach c_i-1(X_i-1)#ID_i-1 in lookup(c_i-1)
   foreach c_i+1(X_i+1)#ID_i+1 in lookup(c_i+1)
      ... 
      if alive(ID_i) and ...
      ...activate(ID_i); ...
      return true
   ...end
```

---

3.3 Basic Compilation Scheme

For now, we only consider programs without negation or priorities (this corresponds to regular CHR programs).

A central concept in our evaluation strategy is the active fact. Each asserted fact is activated once. The active fact goes through all occurrences of its class, searching for applicable rule instances. We perform lazy matching, i.e., each time an applicable rule instance is found, it is fired immediately. This constitutes the fundamental difference with eager matching algorithms such as Rete [5, 39] and TREAT [6, 7], which first compute all applicable instances before selecting one to fire (cf. Section 6).

The activate procedure for a class c with n positive occurrences has the following form:

```plaintext
procedure activate(c(ID, X))
   if occurrence_c_1(ID, X) return...
   if occurrence_c_n(ID, X) return end
```

By default, a class’ occurrences are traversed in a top-down, left-to-right order (later sections refine this order). An occurrence procedure returns **false** if the active fact is still alive after firing all applicable rule instances matching that occurrence, and **true** otherwise.

The compilation scheme for a single occurrence procedure is shown in Fig. 1. Lines 2–7 constitute a nested iteration over all n − 1 join partners, facts that may match the remaining positive patterns. If the active fact fixes some variables, it is said to seed the search for matching partners. A rule instance found is valid if all its facts are alive (line 8) and mutually distinct (line 9), and all guard constraints are satisfied (line 10). After verifying that it
procedure occurrence_pc(L, ID)  
  foreach mem(A, X) #ID in lookup(mem)  
    foreach prog(I, L, B, A) #ID in lookup(prog)  
      if alive(ID) and ... and alive(ID)  
        if allDifferent(ID, ID, ID, ID)  
          if L = L and L = "add" and A = A and B = B  
            if notInHistory(add, ID, ID, ID, ID)  
              addToHistory(add, ID, ID, ID, ID);  
              kill(ID); kill(ID);  
              ID^n = create(mem, A, X+Y);  
              store(ID^n);  
              ID^n = create(pc, L+1);  
              store(ID^n);  
              cleanHistory();  
              activate(ID^n); activate(ID^n);  
            end  
          end  
        end  
      end  
    end  
  end  

 Fig. 2. Naive compilation of the (retracted) pc(L) occurrence of the RAM simulator rule of Ex. 5.

has not fired before (line 11), the newly found instance is fired: the history is updated (line 12), the necessary facts are retracted (line 13), and the RHS is executed (lines 14–18). The call to cleanHistory on line 17 removes all instances containing retracted facts from the history.

Example 6 (Running example): The naive compilation of the pc occurrence of Ex. 5 is shown in Fig. 2. This example is used as a running example later in Section 4.

Note that asserted facts are activated left-to-right, each traversing occurrences in a top-down, left-to-right order. This is in accordance with the so-called refined operational semantics of CHR [33], implemented by most current systems (including [14], [16], [19], [29], [36]). It leads to nice, very intuitive operational behavior. In our opinion, many production systems often exhibit unexpected runtime behavior by not adhering to these simple principles.

A notable difference with CHR’s common semantics is that facts are only activated after every RHS fact is created and stored (lines 14–16 in Fig. 1). In CHR, the RHS is commonly evaluated incrementally: facts are activated before other facts further in the RHS conjunction are asserted. When negation or priorities are added to the language, however, the only sensible semantics employs the here used batch evaluation (see [30], [31]).

On a more technical note: To activate asserted facts (line 18 of Fig. 1), the implicit call stack of the host environment is used. Only after all applicable instances involving an activated fact have fired, the call to activate returns, and control returns to the previously active fact (as in [33]). While this is easy to understand (and implement), there is an important caveat. Since pure rule-based languages have no loop primitives, recursive calls are abundant in rule-based programs. The scheme of Fig. 1 therefore often results in sets of mutually recursive host language procedures. In host environments that do not adequately deal with recursion, this leads to stack overflow issues. While typical Prolog systems for example are designed to deal well with recursion, most Java VM implementations have notoriously limited stack space. In [35], [36], several techniques are worked out to address these issues efficiently. In Section 3.5 we propose an alternative, more elegant solution.

3.4 Dealing with Negation

Adding negation to the basic compilation scheme is relatively straightforward. Only two extensions are required. Firstly, for each negated conjunction, a test of form

\[ \neg\text{occurrence}_r \]

is added between lines 10 and 11 of Fig. 1 (1 ≤ i ≤ m). These negation conditions ensure the fact base does not contain facts matching the negated conjunctions. The basic compilation scheme for these procedures is shown in Fig. 3. The arguments passed are the identifiers and variables of the facts matched in the positive LHS.

Secondly, retracted facts must be activated, since retraction of facts may cause new rule instances to become applicable. Calls to the obvious activate(¬r(X)#ID) procedures are thus added to the RHS evaluation, which in turn use neg_occurrence_c(j,k(ID,X),X) procedures generated for each negative occurrence c_i,j.

The main difference with the positive case (Fig. 1) is that all n positive patterns must be matched. The retracted fact’s arguments can be used to seed the join though.

The basic compilation scheme in this section corresponds closely to the LEAPS algorithm, as described in [8], [15], [40]. The main difference is the way reapplication is prevented. LEAPS uses a complicated technique called shadowing. Essentially, all retracted facts are kept in a so-called shadow fact base, together with a timestamp of when retraction occurred. This allows the algorithm to prevent unwanted reapplication. One obvious drawback is that the shadow fact base is a potential memory leak. It is very hard to determine when shadow facts are no longer necessary (see [8], [15], [40]).

Unlike shadowing, our approach is easily understood. The history of previously applied rule instances readily prevents unwanted reapplication. Removing inapplicable instances after each rule application (line 17 of Fig. 1),
enables the reapplications allowed by the semantics (see Section 2.2.3). It also bounds history size to $O(f^\chi)$, with $f$ the size of the fact base, and $\chi$ the largest number of positive patterns in a LHS. In [8], [41], the LEAPS authors argue that similar bounds can be achieved for shadowing as well, using a complicated filtering algorithm.

For lazy matching algorithms, reapplication prevention thus determines the worst case memory complexity. Fortunately, histories can often be eliminated entirely, as well, using a complicated filtering algorithm.

Section 4.5. For a comparison of the memory complexity of eager and lazy matching: see Section 6.

### 3.5 Dealing with Priorities

Priorities add the restriction that no rule instance may fire if any instance of higher priority is applicable. Consequently, an active fact should only consider an occurrence after all other facts have tried all their occurrences of higher priority. For this, we introduce a last runtime data structure called the schedule (technically a priority queue). Active asserted or retracted facts, denoted $c(\bar{X})\#ID@p$ or $\sim c(\bar{X})\#ID@p$, can be scheduled at a given priority $p$. The schedule supports their efficient retrieval in ascending order of priority. We assume $c(\bar{X})\#ID@p$ items are automatically removed from the schedule when or after the $c(\bar{X})\#ID$ fact is killed.

So when evaluating a RHS, retracted and asserted facts are no longer activated directly, but simply scheduled at their highest priority (i.e., line 18 of Fig. 1 is replaced). After all items are scheduled, the following procedure is called, with $p_{active}$ the priority of the applied rule:

```plaintext
procedure activate( p_{active} )
    while scheduledPriority() ≥ p_{active}
        activate( pollScheduled(), p_{active} )
    end
end
```

It activates all required scheduled facts. When the call to this procedure returns, all applicable instances of strictly higher priority have fired, and the previously active fact can safely resume its search for applicable instances.

The compilation scheme for activating an asserted fact at a given priority is shown in Fig. 4; the scheme for retracted facts is analogous. The $p_{active}$ argument is the highest priority at which another fact is active. Using a `switch` statement to start at the right priority, the active fact keeps trying occurrences until it is either retracted, or until it has to yield control to the previously active fact or to some other fact on the schedule ($scheduledPriority()$ returns the highest scheduled priority, or $-\infty$ if the schedule is empty). The occurrence order is changed to reflect priorities. For occurrences of the same priority, the familiar top-down order is used.

We opted not to consider occurrences of the same priority as a previously active fact (hence the ‘$\leq p_{active}$’ tests in our `activate` procedures). In other words, the following invariant holds: at most one fact is active at any given priority. The rationale for this design choice is:

1. Our `switch` statement’s fall through mechanism ensures that, if no `return` is executed, occurrences of lower priority are tried as well.

Fig. 4. Activation of facts of class $c$, extended to deal with priorities. We assume occurrences $f_i$ to $l_i$ of the class have priority $p_i$ (that is: $f_i = l_i - 1 + 1$), with $p_{i+1} < p_i$.

1. It eliminates the call stack issues raised in Section 3.3, as all recursive calls necessarily go through the schedule. For firing higher priority rules, however, the implicit call stack is still used. This is typically more efficient, and certainly easier to implement, than using e.g. a self-maintained stack (see [35], [36] for more information).
2. The above invariant facilitates static analysis required for certain optimizations introduced in Section 4 (particularly late indexing; see Section 4.3.1).

### 4 PROGRAM ANALYSIS AND OPTIMIZATION

#### 4.1 Class Invariants

We begin by introducing two class invariants that will prove invaluable to several advanced optimizations.

##### 4.1.1 Set semantics

Often, set semantics is desired rather than our default multiset semantics. Not only does it facilitate more efficient data structures, allowing identical facts may even affect a program’s runtime complexity or termination.

**Example 7:** A nice example is the `edge_label` class of the classic production rules program WALTZDB [9], an extension of the WALTZ program of Ex. 1. Under multiset semantics, this program does not terminate, as infinitely many duplicate `edge_label` facts are derived.

To resolve such issues, we propose a set annotation for class declarations. For our example, this gives:

```plaintext
class edge_label( int p, ..., int 1_id ) :: set
```

Each time a fact of a class with set semantics is asserted, the fact base is first checked for syntactically identical facts. If present, the newly asserted fact is killed.

Unavoidably, checking for duplicates involves a runtime overhead, and typically also additional fact indexing (see Section 4.2.4). Often, however, the programmer knows the program never asserts duplicate facts. To state
this invariant, we propose a \texttt{set} declaration (see Ex. 8 below). While not affecting the operational semantics, this knowledge can be exploited by the compiler. These annotations are also invaluable as part of a program’s documentation, and could even be verified automatically if desired, either when running the program in some debugging mode, or by static program verification.

4.1.2 Functional dependencies
A second, very important class invariant is the existence of functional dependencies between fields. Functional dependencies are an established concept in relational databases. For our purpose, we defined functional dependencies as follows: a set of fields $X$ of a class $c$ is said to functionally determine a set of fields $Y$ of that class if the following runtime invariant holds: if the fact base contains a $c$ fact with the $X$ and $Y$ fields bound to values $V$ and $W$, then for any (other) $c$ fact in that fact base with fields $X$ bound to $V$, the $Y$ fields are bound to $W$. Such a functional dependency is noted $X \rightarrow Y$. We further require all functional dependencies to be non-trivial, that is: $X$ and $Y$ must be disjoint, and $Y$ non-empty.

Example 8: The classes of the RAM program of Ex. 5 may have following declarations:

$$
class \text{mem}(\text{int \ addr, int \ val}) :: \text{set, fd(\text{addr} \rightarrow \text{val})}
$$

$$
class \text{prog}(\text{int \ label}, ...) :: \text{set, fd(\text{label} \rightarrow _{\text{}})}
$$

$$
class \text{pc}(\text{int \ label}) :: \text{set, fd(\text{\text{label} \rightarrow _{\text{}}})}
$$

Clearly, each mem memory cell associates a memory address with a single value. This functional dependency is called total, as the addr field functionally determines all fields. As total dependencies are very common, we propose the ‘\_’ shorthand notation to denote ‘all other fields’. This is illustrated by the declaration of prog. The pc class is special, as there is never more than one pc fact present. We call such a class a singleton class.

Example 9: As discussed before in Ex. 3, another example is the edge class in the WALTZ program:

$$
class \text{edge}(\text{int \ p1, int \ p2}, ...) :: \text{set, fd(p1,p2 \rightarrow _{\text{}})}
$$

As illustrated by these examples, functional dependencies mostly co-occur with set semantics. Shorthand declarations such as set $(X \rightarrow _{\text{}})$ and key $(X)$ (in analogy to relational databases) are therefore supported.

Knowing about functional dependencies allows a compiler to improve, among other things, indexing (Section 4.2.5) and join ordering (Section 4.2.6).

4.1.3 Deriving class invariants
Most CHR systems do not support invariant declarations such as those proposed above. Instead, it is common practice in CHR programming to specify class invariants by adding appropriate rules. As these rules should fire before any other rule, these idioms are further facilitated by the advent of priorities (see also [31]).

Example 10: The set semantics of the \texttt{edge\_label} class of Ex. 7 for instance, would be enforced as follows\footnote{ Actually, CHR does not support the explicit use of timestamps. Due to CHR’s common operational semantics though, the guard may be omitted without affecting the rule’s semantics (see Section 4.4.4).}

$$
\text{ss} @ 100 :: +\text{edge\_label} (P_1, P_2, \text{Name}, \text{Id}) \# t_1,
-\text{edge\_label} (P_1, P_2, \text{Name}, \text{Id}) \# t_2, \ ?(t_1 < t_2).
$$

In fact, invariants may even be derived from rules not added for the sole purpose of removing duplicates:

Example 11: The following rule from the DIJKSTRA program of [42] implies min is a singleton class:

$$
\text{keep\_min} @ 10 :: +\text{min}(\_, A), -\text{min}(\_, B), \ ?(A < B).
$$

From such rules, optimizing CHR compilers are able to derive the existence of runtime invariants. For detailed descriptions of algorithms used, we refer to [17], [18].

4.2 Optimizing Join Computation
The most critical part of any rule-based system is the search for matching join partners to form rule instances. This section surveys several techniques to optimize join computation. While we explain and illustrate them here for the positive part only (Fig. 1 extended with negation conditions), they equally apply to the computation of the joins for negated conjunctions (Fig. 3).

4.2.1 Backjumping
A fact in lines 1–7 of Fig. 1 (i.e., the active fact or some join partner) needs a test (a guard or a negation condition) if it binds variables required by that test.

Example 12: The following LHS occurs in WALTZ:

$$
\text{initial\_boundary\_junction\_L} ::
-\text{stage}("\text{find initial boundary}") ,
+\text{junction}(P_1, P_2, _{\text{Base}}, "L") ,
-\text{edge}(\text{Base}, P_1, Q_1, _{\text{Plotted}}) ,
-\text{edge}(\text{Base}, P_2, Q_2, _{\text{Plotted}}) ,
\sim(\text{junction}(\_, \_, \_, \_, B, _{\text{}}), ?B > Base) \Rightarrow \ldots
$$

Suppose that the partners of an active stage fact are matched in the order they appear in the LHS (cf. Section 4.2.6). Then the only fact seeding the negation condition is the one matched by the junction pattern. Therefore, when the negation condition fails, resuming the nested iteration over edge facts is pointless. Instead, the next junction fact should be tried.

The backjumping optimization ensures that when a test fails, a jump is executed to resume the inner-most seeding loop (see also Section 4.2.5). If no such loop exists, true is returned. Without backjumping, a phenomenon called 

\text{trashing} occurs, where inner loops are iterated exhaustively\footnote{In the case of Ex. 12, with proper indexing (Sections 4.2.4–4.2.5), trashing would actually be limited, as by functional dependency only a single matching edge fact exists for each inner loop (as seen in Ex. 9).}, only to find the test always fails.

4.2.2 Loop-invariant code motion
All tests, particularly identifier comparisons and most guards, should normally be performed as early as possible. Not doing so, leads to 

\text{trashing} (as in Section 4.2.1).

Example 13: The improved compilation of the RAM simulator example introduced in the previous section is listed in Fig 5. Scheduling the ‘L = L1’ on line 4 for instance avoids enumerating all mem $(B_1, X)$ memory cells before the right program instruction is found.
Fig. 5. Optimized compilation of the RAM simulator example of Fig. 2 after loop-invariant code motion.

If after the RHS’s execution the search for join partners is resumed, hoisting a test is only correct if that RHS cannot change its outcome. Otherwise, inner loops may cause rule instances to fire for which these tests no longer hold. Most standard guards (called monotonic guards in CHR literature) can safely be moved, but this is not necessarily true for liveness tests or negation conditions:

Liveness tests: Resuming search after a fact matched by an outer loop is retracted—by the rule itself or indirectly by some activated fact—is a clear source of trashng. But hoisting all liveness tests (line 8 of Fig. 1) into the surrounding loops is incorrect in general, as rule applications can change their outcome. The solution is to move them after line 19 of Fig. 1, that is: after RHS evaluation. The outermost loop’s fact is tested first, and so on. If one of the partners tests dead, a jump is used to the corresponding iteration. Of course, if the rule itself retracts some partner, the jump becomes unconditional.

Negation conditions: Similarly, a negation condition may be hoisted into outer loops as long as it is retested after each RHS execution. If facts matching the negated conjunction are then found, a jump to the inner-most seeding loop is performed (cf. Sections 4.2.1 and 4.2.5), or true is returned if no such loop exists.

Unlike liveness tests and most guards, testing negation conditions can be expensive. It therefore pays to detect, using for instance abstract interpretation [20] whether the RHS either never or always adds facts matching the condition. In the former case, the test can safely be moved, without retesting. In the latter, the backjump becomes unconditional. Note further that moving an expensive test (negation condition or an expensive guard) to an outer loop may increase the number of times it is called, thus actually decreasing performance. Heuristics are used to decide whether or not to move such tests.

4.2.3 Non-robust iterators

Due to the highly dynamic nature of the fact base, the robustness property of iterators (see Section 3.1) is hard to implement, and often has a significant performance penalty. This property is not always required though:

1) If after the execution of the RHS the iterator is never resumed (if the active fact is retracted, or due to an unconditional jump to a more outer loop).

4. Search is not resumed if the active fact is retracted, but also for instance if none of the lookups return an iterator: see Section 4.2.5.

2) If static analysis shows the RHS is guaranteed not to modify the relevant part of the fact store.

3) None of the iterators used to test negated conditions (Fig. 3) have to be robust.

Non-robust iterators are used where possible because they can typically be implemented more efficiently.

Example 14: All iterators in the RAM running example may be non-robust as the rule retracts the active fact.

4.2.4 Fact indexing

Efficient, selective lookup of candidate join partners is imperative. Therefore, indexes on the fact base are used.

Example 15: In Fig. 5, line 3 iterates over all prog facts, each time immediately testing the ‘L1 = L’ guard. There is however at most one prog fact with given label L, as implied by its class invariants (see Section 4.1). Retrieving this single fact using an appropriate index reduces the linear time complexity of this part of the join computation to constant time. A similar reasoning applies to the lookup of the mem fact (lines 5–6 of Fig. 5).

For lookups of join partners via one or more known arguments, tree-, hash-, or array-based indexes are used [16], [18], [19], [42]. Tree-based indexes cannot only be used for equality-based lookups, but also for pruning the join partner search space in case of inequality guards [16]. The other two types are particularly interesting as they offer (amortized) constant time operations. In [25], an alternative indexing technique is introduced based on an improved internal representation of field values.

Specialized lookup operations are used to retrieve facts satisfying given guard conditions. While indexes substantially improve performance, each index also involves a non-negligible maintenance cost. Heuristics are therefore used to decide which indexes to build.

4.2.5 Exploiting class invariants

If (derived) class invariants imply a lookup returns at most one fact, more efficient index structures can be used, and specialized lookup routines that return a single fact instead of an iterator [17], [18]. We denote such specialized procedures lookup_s (‘s’ for single).

Example 16: Fig. 6 shows the optimized compilation of our running example. If the specialized lookup_s operations on lines 3–4 use array- or hash-based indexing, both partners are found in $O(1)$ time, instead of $O(p \times m)$ (with $p$ the number of RAM program instructions, and $m$ the number of used memory cells). Also,
the functional dependency of the \textit{prog} class was used to select proper indexing (line 3). If a compiler does not know about this invariant, it typically adds an extra index on the combination of the first, second and fourth argument. This clearly shows functional dependencies can considerably improve both space and time performance by reducing the number of indexes maintained.

Note that replacing iterations with simple lookups affects the optimizations discussed in Sections 4.2.1–4.2.2. Any backjump to such an optimized lookup must instead go to the loop immediately surrounding it, if any.

4.2.6 Join ordering

Time complexity is often determined by the join order—the order in which join partners are looked up (so far, this order was based on the rule’s normal form). It determines the earliest position where guards, and thus indexes, may be used. The general goal behind join ordering is to maximize the usage of indexes, in order to minimize the number of join partners tried. The optimal join order may depend on dynamic properties, such as the size of the fact base. If no functional dependencies are declared or derived, a compiler must rely on heuristics to determine the join order. The join ordering problem is NP complete. The most comprehensive treatment of join ordering for CHR is [23].

\textbf{Example 17:} Line 2 of Fig. 6 iterates over all \textit{mem} facts. Lacking any information on \textit{A} (or \textit{L}), no index can be used. Using the join order depicted in Fig. 7, however, all lookups become optimal (provided proper indexing is used). For determining this join order, functional dependencies are again indispensable (see [23]).

### 4.3 Reducing Fact Base Overhead

Indexes are fundamental for the efficient retrieval of candidate join partners. Each index, however, increases the cost of the \textit{store} and \textit{kill} operations. In Section 4.2.5, we already saw how functional dependencies help reduce the number of indexes. Here, we introduce two more optimizations to reduce fact base overhead.

4.3.1 Late indexing

So far, facts are stored immediately after being created. A fact’s lifetime, however, is often very short. Frequently, a fact is even killed shortly after activation. The goal of late storage [16], [18], [19], [20] is to defer storing facts as long as possible. In many cases the fact will then be killed before it is stored. This avoids the considerable overhead of fact base additions and removals. The performance gain is particularly significant if indexes are used. If index operations are non-constant (e.g. tree indexes [16], [18]), late indexing may even improve runtime complexity.

Late indexing [31] goes beyond late storage by selectively adding facts to required subsets of indexes only, instead of all indexes at once. The approach we outline here further refines that of [31], and allows fact indexing to be postponed (and hence avoided) even more.

Facts are only added to those indexes through which they may be observed. A fact \( f \) is observed through an index \( I \) if, before execution control returns to \( f \), any active fact (asserted or retracted) causes \( f \) to be looked up using \( I \)—either as a positive join partner, or while testing a negation condition. The static program analysis that determines when and where facts must be stored is called the observation analysis. It typically uses abstract interpretation. For late storage, this is worked out in [20].

Concretely, late indexing postpones the \textit{store} operations of Fig. 1, lines 14–16. Facts are only stored there in indexes through which they may be observed by negation conditions before they are activated. But normally, a fact is only indexed while it is active and about to relinquish control in one of these three cases:

1) prior to the execution of a RHS
2) when yielding control at a transition to a lower priority (see Section 3.5, Fig. 4, lines 9–11)
3) after all occurrences have been traversed.

In the first case, the initially activated facts are obvious. The invariant established in Section 3.5 further ensures only rules of strictly higher priority are considered before the previously active fact regains control. In case of a transition from priority \( p_i \) to \( p_{i+1} \), any fact scheduled between \( p_i \) and \( p_{i+1} \) may be activated\(^5\), and only occurrences of priority higher or equal to \( p_{i+1} \) are considered. From this, the observation analysis can accurately determine the required indexing operations.

Note that late indexing may change the order in which rules fire\(^6\), as active facts do not observe not-yet-indexed facts. All applicable instances are still found though when these facts are activated themselves.

4.3.2 In-place modifications

Very often, shortly after a fact is retracted, a new, only slightly different fact is asserted. Most production systems even offer syntactic sugar to perform such modifications. The \textit{in-place modifications} optimization [22] reuses the original fact’s representation, simply assigning a new identifier, and overwriting the modified fields. Affected indexes have to be updated also, but indexes independent of the modified fields require no update.

---

\(^{5}\) Actually, asserted facts scheduled at \( p_{i+1} \) do not have to observe the previously active fact, as that fact will also still be activated at \( p_{i+1} \).

\(^{6}\) Changing the rule order may affect performance, either positively or negatively. This effect can typically be controlled using priorities.


Example 18: Both facts asserted by the copy rule of our running example are modifications of retracted facts. So lines 11–14 of Fig. 2 can be replaced for instance by:

\[
\begin{align*}
\text{id}^1 &= \text{modify}(\text{id}_2, \{\text{value} = X+Y\}) \\
\text{id}^2 &= \text{modify}(\text{id}_1, \{\text{label} = L+1\})
\end{align*}
\]

As no index exists on the value field of mem, the first operation no longer requires updates to indexes.

4.4 Optimizing Fact Activation

4.4.1 Reducing schedule overhead

As with fact base operations, redundant calls of schedule operations (see Section 3.5) should be avoided where possible. In [31], three optimizations are proposed. Firstly, if firing a rule is known never to activate facts of higher priority, the call to activate(p) can clearly be omitted. This call can also be omitted if the active fact is retracted, as control always returns to a loop in activate(p), which already checks the schedule for more scheduled activations. Lastly, facts can be activated directly, i.e. without going through the schedule, if their priority is known to be higher than that of the activating rule, and than that of all other facts activated by that rule.

4.4.2 Passive modification

By default, a retracted fact is activated to find additional applicable rule instances. As seen in Section 4.3.2, however, retracted facts are often immediately replaced by only slightly modified instances. If no negative occurrences have guards (implicit or explicit) on the modified fields, activating the retracted fact is unnecessary.

Example 19: The retraction of the edge facts in the make_L rule of Ex. 2 can be done passively. No negative occurrence of edge considers the modified join field.

Passive modifications avoid superfluous traversals of negative occurrences, lead to more passive occurrences (Section 4.4.3), and improve the results of static program analyses (e.g. the observation analysis of Section 4.3.1).

4.4.3 Passive occurrences

An occurrence is passive if static analysis shows that the corresponding rule can or must not fire with the active fact matching it. Passive occurrences are simply skipped by active facts. Detecting passive occurrences is very important, not only because superfluous searches for join partners are avoided, but also because index structures only required for these searches can be discarded.

Subsumption: An occurrence is subsumed by another occurrence if each fact that matches the former pattern also matches the latter, taking into account join partners, negation conditions and guards. If an occurrence is subsumed by another occurrence in the same rule, and that rule retracts at least one fact, the former occurrence can be made passive. If an occurrence is subsumed by occurrences in earlier rules, subsumption analysis may also detect redundant rules (often indications of programming mistakes). For more information: see [21].

Example 20: Both Ex. 2 and Ex. 12 contain a rule with two edge occurrences that subsume each other. In both cases, one of these occurrences can thus be made passive.

Never retracted: If facts matching a negative occurrence are never retracted actively (see Section 4.4.2), that occurrence can be made passive.

Never stored optimization: Certain common idioms in rule-based programming involve classes whose facts are always retracted at some point. First some examples:

Example 21: The RAM program contains a rule:

```
fallback @ -100 :: -pc(_) \Rightarrow fail.
```

If fires only if the simulated RAM program reaches an illegal state (fail halts execution in a failed state). Such an unconditional retract at a lower priority occurs frequently, and is similar to a default case in switch statements of modern imperative programming languages.

Example 22: In the WALTZ program, each rule involves a stage pattern. When all rules of some stage have fired, rules such as the following move to a next stage:

```
done_plotting @ -10 ::
	-\text{stage}("\text{plot remaining edges}") \Rightarrow \text{stage}("\text{done}").
```

In both cases, facts are always retracted at some lower priority. As the following example will help elucidate, always retracted facts may lead to passive occurrences.

Example 23: In the RAM program, all occurrences of the pc class are retracted (all rules resemble the add rule of Ex. 5). Combined with the fallback rule of Ex. 21, this suggests pc facts should never be stored. If pc facts are made never stored, non-pc occurrences can never lead to complete rule instances, and can all be made passive. This improves performance considerably: mem facts are never activated, less indexes are built for prog facts, etc.

In general, for this optimization to work, facts must be indexed more eagerly (mem and prog facts in the example) to ensure that the never stored active facts observe their passive partners. The trade-off with late indexing (Section 4.3.1) is made through heuristics. 7

Always retracted: If a fact matching a negated condition is always retracted at some higher priority, the tests for these negated conditions can be omitted.

4.4.4 Retraction preference

CHR compilers commonly use the following heuristic: when a single rule contains multiple occurrences of the same class, an active fact tries the retracted occurrences of that rule first (see also [33]). Retracting the active fact rather than facts already stored can considerably improve performance (see late indexing, Section 4.3.1), or even avoid non-termination (see Ex. 10, Section 4.1.3).

4.5 Optimized Reapplication Prevention

Despite the efficient implementation techniques and data structures used by most CHR systems (see [24] for an overview), maintaining a history remains relatively expensive. For most rules, however, the history can be

7. For CHR without priorities or batch evaluation this optimization, the never stored optimization is more straightforward [16], [18].
eliminated without affecting the operational semantics. A first, obvious optimization is not to add instances of rules that retract at least one fact. For the remaining rules, [24] introduces three optimization techniques:

1) Observation analysis: late indexing—particularly if combined with the recursion-breaking invariant of Section 3.5—often prevents join partners from observing a fact before it is activated (see Section 4.3.1). Checking a history is then not necessary. If this holds for all active occurrences of a rule without active negative occurrences, its history can be eliminated entirely.

2) Timestamps: for negation-free rules, it suffices that the active fact’s timestamp is greater than that of all join partners (this technique is independently proposed in [8], [15], [40]). Added advantage is that the resulting timestamp comparisons can be hoisted to outer loops (cf. Section 4.2.2), even for rules that retract facts. Extending this principle to negation, without maintaining a shadow fact base (cf. Section 3.4), is impossible in general. 8

3) Idempotence: practice shows reapplication prevention is often more expensive than reapplication. Of course, discarding a rule’s history is only correct if this affects neither termination nor the final fact base. We call such rules idempotent rules. Intuitively, a rule is idempotent if firing an instance that has been fired before has no observable effect. For more details: see [24].

4.6 Program Specialization

Static analysis sometimes shows the program can be improved. Doing so frequently facilitates further optimizations, or enhance results from other static analyses.

4.6.1 Class specialization

In [26], source-to-source transformations are proposed to specialize classes and rules based on manifest field values in LHS patterns. These specializations improve the accuracy of static program analyses, fact indexing, and occurrence dispatch (see also Section 7.1.3).

4.6.2 Guard simplification

For each occurrence, guard simplification looks at retracted occurrences earlier in the active fact’s occurrence order to remove redundant guards, thus also facilitating other analyses such as passive occurrence detection. Guard simplification is best described in [21].

Example 24: The guard simplification principle can be extended to negation conditions. The following excerpt, adapted from the Dijkstra program of [42], illustrates a very common programming idiom with negation:

\[
\text{scanned} :: \neg \text{relabel}(N), +\text{dist}(N, _). \\
\text{not_scanned} :: \neg \text{relabel}(N), \sim \text{dist}(N, _) \Rightarrow \ldots
\]

No matter which occurrence an active relabel fact considers first, if the corresponding rule does not fire, the other rule is always applicable, and the second dist lookup can be omitted. It also becomes apparent that the relabel class is never stored (see Section 4.4.3). 9

5 Empirical Evaluation

Using the principles outlined in Sections 2 to 4, we developed a new Java-based compiler, designed to be the successor of the JCHR system [28]. We compared against two established production rule engines. Clips [2] was chosen as a reference, because a recent performance survey suggests it is currently the most efficient system available [43]. Jess [44] is a more recent Java-based implementation. The results measured are as follows: 9

![Fig. 8. Performance comparison for two famous benchmarks.](image)

<table>
<thead>
<tr>
<th>Benchmark [origin]</th>
<th>Clips 6.30</th>
<th>Jess 7.1p2</th>
<th>JCHR 2.0α</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra(8192) [42]</td>
<td>16.3 34</td>
<td>22.2 (136%) 25</td>
<td>1.24 (7.60%) 447</td>
</tr>
<tr>
<td>primes(2048) [27]</td>
<td>2.02 1.88</td>
<td>3.59 (178%) 1.05</td>
<td>0.07 (3.46%) 54.2</td>
</tr>
<tr>
<td>RAM_fib(50k) [38]</td>
<td>12.2 45.1</td>
<td>36.1 (296%) 15.2</td>
<td>0.69 (5.62%) 803</td>
</tr>
<tr>
<td>union(25k) [9]</td>
<td>40.7 5.5</td>
<td>10.4 (25.6%) 21.6</td>
<td>0.28 (0.70%) 793</td>
</tr>
<tr>
<td>Manners(256) [9]</td>
<td>48.6 0.69</td>
<td>361 (742%) 0.09</td>
<td>0.19 (0.39%) 202</td>
</tr>
<tr>
<td>sudoku lcp-17 [43]</td>
<td>0.85 9.39</td>
<td>4.37 (539%) 1.78</td>
<td>0.44 (52.3%) 19.8</td>
</tr>
<tr>
<td>lcp-17 (annex)</td>
<td>3.06 3.2</td>
<td>54 (1754%) 0.18</td>
<td>1.10 (36%) 8.96</td>
</tr>
<tr>
<td>Waltz(100) [9]</td>
<td>6.25 4.47</td>
<td>32.1 (513%) 0.87</td>
<td>1.12 (18%) 24.9</td>
</tr>
<tr>
<td>WaltzDB(32) [9]</td>
<td>2.8 9.52</td>
<td>12.9 (461%) 2.01</td>
<td>0.42 (15.1%) 53.7</td>
</tr>
<tr>
<td>wordgame(200) [44]</td>
<td>4.49 4.55</td>
<td>5.51 (123%) 3.72</td>
<td>2.92 (65.1%) 6.98</td>
</tr>
</tbody>
</table>

For each system, a first column gives the average running time in seconds (between parentheses is the relative performance compared to Clips), and a second the average rule application rate in kRAPS (1,000 Rule Applications Per Second). We used standard CHR and production rule benchmarks only, without optimizing the programs for either system. The three systems ran equivalent programs (using automatic source-to-source transformation mostly), and fired exactly the same rules. Only for the non-deterministic WaltzDB benchmark, the three systems fired slightly different sets of rules.

The Clips-to-JCHR transformation used for the last six benchmarks annotates all classes with set (see Section 4.1). By manually adding better invariant declarations, further performance gains were possible: Manners(256) for example ran in 70ms, Waltz(100) in 617ms (about 3 and 2 times faster resp.). This clearly shows the importance of invariant annotations (and late indexing).

Our prototype outperformed state-of-the-art production rule engines, often by several orders of magnitude.

9. Benchmarks were performed on an Intel®Pentium®4 CPU 2.80GHz with 1GB of RAM running Linux 2.6.22. Java code was compiled using Eclipse SDK 3.4.2 and run using HotSpot™ JRE 1.6.0.
Admittedly, this comparison should be taken with a grain of salt, as JCHR is the only system that compiles its programs. But, on the other hand, only Clips is implemented in C, which typically is more efficient: see [29]. One could therefore argue that the comparison between JCHR and Clips is reasonably fair.

More importantly though: JCHR mostly scales better with larger problem sizes, often with better asymptotic time complexity. This is clearly shown in Fig. 8. These results are in line with earlier findings [9], [10].

Finally, while measuring and comparing actual memory usage of different systems is hard, our experiments did confirm space complexity of lazy evaluation is superior as well. We discuss space complexity in Section 6.

5.1 Case Study: Misuse Detection

In [45], an in-depth analysis is presented why Rete’s performance is lacking in the area of misuse detection. Its critiques on Rete coincide precisely with the principal rationale behind lazy matching algorithms (see Section 6). We therefore repeated their experiments, and measured the following timings when processing 10,900 events:

- Clips 6.30s: 3.2s
- SAM: 1.3s (40.5%)
- CCHR: 0.28s (8.9%)

SAM is a special-purpose misuse detection algorithm [45]. We used CCHR [29] because it was easier to link with the pre-existing C++ code. Clips and CCHR used completely equivalent rule programs. We found that CCHR outperformed not only Clips, but also the special-purpose SAM system. This result indicates lazy matching is indeed better suited than Rete for performance intensive applications such as misuse detection.

6 Discussion and Related Work

As seen in Section 3.4, our lazy evaluation scheme is closely related to LEAPS. First introduced in [8], [15], LEAPS served as the basis for the Clips++ system [10], and its successor Venus [46]. Aside from the high-level reconstruction of [40], only little has been published on their implementation (apparent exception is [47], where the importance of fact indexing is stressed). No clear description of how to incorporate priorities, or how to optimize the basic algorithm has ever been published.

On possible optimizations of Rete, on the other hand, there is a vast literature. Space limitations prohibit us from covering all related work in detail. For an excellent introduction of Rete and its most common optimizations, we refer to [39]. For related work on specific optimizations, we also refer to the related work sections of the many publications cited in earlier sections.

Fundamentally, Rete differs from LEAPS and our approach for the following two reasons:

1. Rete is an eager matching algorithm, and
2. Rete performs join indexing

Practice shows these properties, while surely beneficial in certain specific cases, mostly lead to poor space and time performance (cf. our empirical evaluation in Section 5). We now discuss both properties in more detail.

Eager matching algorithms maintain the set of all applicable rule instances, called the conflict set. After each fact assertion or retraction, this set is updated, before selecting a next instance to fire (conflict resolution [34]). As nicely shown in [8], numerous thus computed rule instances typically never fire, because they soon become inapplicable again. Lazy matching does not waste time computing these superfluous instances [8], [9].

Because rules often have overlapping LHSs, naive evaluation computes the same (partial) joins many times. The idea of join indexing is to compute them only once, storing them for later reuse (Rete indexes all partial joins computed in so-called beta memories). As with all forms of indexing, however, the time gained by reusing joins should be weighed against the inherent cost: if indexed joins are rarely reused, or join partners are frequently retracted, join indexing only decreases space and time performance. Most studies indeed confirm that Rete’s injudicious join indexing strategy mostly works counterproductive [6], [7], [8], [9], [10], [11].

TREAT is a matching algorithm based on exactly that observation [6], [7]. While TREAT remains an eager matching algorithm, it differs from Rete mainly by never performing join indexing. Another property TREAT has in common with lazy algorithms is that, without the fixed network of beta nodes and memories of Rete, TREAT also has the freedom to properly seed join ordering by a form of active fact. As shown in [6], [7], this is a significant factor in its improved performance.

Clearly, the space complexity of both eager matching algorithms is $O(f^k)$, exactly the same as the complexity derived in Section 3.4 for our approach. In practice though, the space behavior of lazy matching algorithms is mostly superior. Indeed: in Section 4.5, we argued that most practical rules and programs do not require a history. Therefore, typical programs use only $O(f)$ space.

For improved performance, several recent rule engines provide a so-called sequential matching algorithm. From a given fact base, this algorithm first computes the conflict set, and then fires all computed instances, but without activating facts (i.e., without inserting assertions and retractions in the Rete network). This seems a rather ad hoc workaround to the inherent overhead of the Rete network. Clearly, lazy matching is a much cleaner, more satisfactory solution to Rete’s performance issues.

6.1 Beyond the Core Language

While our core language is already very expressive [38], and surely sufficient for many applications [13], current Rete-based production systems typically offer many more features. Most of them, however, should not be construed as being peculiar to Rete, and can be added to lazy evaluation schemes as well. Aggregates (including nested and user-defined aggregates) for instance can be added using source-to-source transformations [48], dynamic priorities are addressed by [31], and efficient truth maintenance techniques are studied in [49]. Lazy
matching is also known to facilitate database integration [8], [9]. Dynamically asserting rules is admittedly difficult using the static compilation techniques currently used by CHR systems. However, the same techniques could readily be used for a more dynamic interpreter-based implementation (see also Section 7.1.2). Only in the pathological case where conflict resolution requires the full conflict set, lazy matching cannot be used.

7 CONCLUSIONS AND ONGOING WORK

We clearly described how to design a powerful, highly optimized, lazy rule engine, building further on the wide spectrum of novel analyses and optimizations recently produced by the CHR community. With this extensive survey, we hope to revive interest in lazy matching techniques, and to promote interchange of ideas between the fields of CHR and production rules.

Our empirical study again shows that for many problems and practical applications, a lazy approach consistently outperforms Rete-based systems. Given the recent surge in adoption of rule based technology, reliable, scalable performance is rapidly becoming more important than ever. We therefore strongly urge that production rule systems incorporate where possible the principles and optimizations presented in this article.

7.1 Ongoing Work

A close examination of the experiments in Section 5 revealed there is still considerable room for improvement. This section lists some promising research directions we are currently pursuing. That some of the current bottlenecks correspond precisely to Rete’s strong suits only further stipulates the importance of cross-fertilization.

7.1.1 Improved indexing

**Join indexing**: As far as we know, lazy evaluation has never been combined with join indexing (see Section 6). Judicious application of join indexing would nevertheless further improve lazy matching performance. Static or dynamic techniques have to be investigated to determine when to use join indexing (see also Section 7.1.2).

**Extrema lookups**: The following idiom occurs regularly in rule-based programs, particularly if aggregates are dynamic or dynamic priorities are not supported:

*Example 25*: The negated condition in the rule of Ex. 12 implies the rule is only applicable for junction facts whose base point field is minimal. These facts are found though by naively iterating over all junction facts, each time testing the negated condition.

Using appropriate data structures to retrieve facts with extreme field values—instead of naive, linear search—would clearly improve time complexity.

7.1.2 Dynamic optimizations

CHR research so far has mainly focused on the static analysis and optimization of programs. Where static techniques fail, dynamic optimizations should be used:

**Dynamic passive occurrences**: Static analysis is not always capable to detect all passive occurrences. Also, in larger programs, the applicability of subsets of rules may change dynamically over time. It is often possible to detect these *dynamic passive occurrences* at runtime. Indeed: all occurrences that employ at least one empty (join) index can be skipped. Such tests are mostly cheap, and may save much redundant matching.

Similar ideas have been proposed in production rule literature. In [40], it is called *active rule optimization*. The Rete equivalent, *unlinking*, is introduced in [39]. It works by dynamically removing edges in the Rete network.

**Dynamic indexing**: As discussed earlier, both fact and join indexing inherently involve non-negligible maintenance overhead. Statically deciding which indexes are used is hard. Possible dynamic optimization techniques include the dynamic adding or removing of indexes, or on-demand index population. Related ideas are proposed in [11], in the context of a Rete-style matcher.

**Dynamic join ordering**: Unlike [7], we strongly believe static join ordering alone is insufficient [23]. Functional dependencies help, but statically determining the correct join ordering remains a problem. Getting the join order wrong often worsens time complexity.

7.1.3 Global optimizations

CHR compilers currently focus on *locally* optimizing the join computation of individual occurrences. For larger programs, more global optimizations are in order:

**Optimized occurrence dispatch**: By default, an active fact linearly traverses all occurrences of its class, even when only a small subset may lead to rule instances.

*Example 26*: A typical example is the *stage* class in the WALTZ program (see Examples 2, 12 and 22). Depending on its single argument, only a very restricted subset of the occurrences should be considered.

For this example, using switch statements significantly improves performance. In general, the *activate* procedure of Fig. 4 should where possible incorporate guards or negated conditions to improve occurrence dispatch. This is clearly an area where we should profit from Rete and TREAT research (these algorithms perform occurrence dispatch using a so-called *alpha network*).

**Inlining and merging occurrence procedures**: Several occurrences (of the same priority) often compute similar joins. Their procedures should then be inlined and subsequently merged to produce more efficient code.

*Example 27*: Most rules in the RAM program have the same form as the *add* rule used in our running example. For each occurrence, an active *pc* fact first retrieves a *prog* fact. This lookup should be done only once. Next, a switch statement on its instruction field could single out which rule is applicable in constant time.

Clearly, such optimizations go well beyond simple occurrence dispatch. We believe them to be an important next step in the state-of-the-art of efficient rule execution.