Sous Projet 1

RETE et réécriture

Production Systems and Rewrite Systems

**Description**: This report studies the relationship between systems and term rewrite systems. Giving a brief definition of both, then comparing all the concepts related to them, their similarities and differences, comparing also the life cycle of each system for finally to study ways for translating one to another.

**Auteur(s)**: Horatiu Cirstea, Claude Kirchner, Michael Moosse, Pierre-Etienne Moreau

**Référence**: MANIFICO/Sous Projet 1/Fourniture 1.2/V1.0

**Date**: 15 novembre 2004

**Statut**: validé

**Version**: 1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de l'économie, des finances et de l'industrie
ILOG, INRIA Lorraine, INRIA Rocquencourt
Historique

31 août 2004 V 0.8 création du document
15 novembre 2004 V 1.0 version finale

Contents

1 Introduction
   1.1 Production Systems .................................. 3
       1.1.1 Informal presentation of production systems .......... 3
       1.1.2 Formal Description of Production Systems ............ 4
       1.2 Rewrite Systems .................................. 7

2 Concepts Comparison
   2.1 Terms ............................................. 7
   2.2 Patterns .......................................... 7
   2.3 Conditions ........................................ 7
   2.4 Matching .......................................... 7
   2.5 Rules ............................................ 7
   2.6 Strategies ....................................... 8
   2.7 Theories ......................................... 8

3 Translation
   3.1 Production Systems into Rewrite Systems .................. 8
       3.1.1 Snyder and Schmolze's approach ....................... 8
       3.1.2 Using an Matching Constraint Approach .............. 10
   3.2 Rewrite Systems into Production Systems ................. 10

4 Examples
   4.1 Abstract Language ................................... 13
   4.2 OPS5 ............................................... 13
   4.3 CLIPS ............................................. 15
   4.4 JeOPS ............................................. 16
   4.5 JRules, using TRL .................................. 18
   4.6 ELAN .............................................. 19
   4.7 JTom .............................................. 20
   4.8 Ac-Rewriting ...................................... 22
       4.8.1 Snyder and Schmolze's approach .................... 22
       4.8.2 Matching Constraint Approach ...................... 23

5 Benchmark
   5.1 A more demanding benchmark ............................ 25
Introduction

There is a strong and somewhat renewed interest in production and business rule systems. For example it is significant that in April 2004 Gardner published a comparative study of the different business rule systems and vendors and estimated that the current market of 200 millions of dollars was promised to a sustained increased for several years.

Production systems, as we will call them from now on, are widely appreciated in industry for their agility, reactivity and flexibility. But putting them in action is also delicate since their semantics is unclear or even unknown and therefore make the development of large systems quite depend of test and experience.

Term rewriting is a different kind a rule based system that emerged from automated deduction and semantics of programming languages in the early seventies. It is now a pretty well understood concept and many implementations show their usefulness, robustness and efficiency.

This report intends to bridge these a priori different views of rule based systems.

Outline of this report:

1 Introduction

1.1 Production Systems
This section recalls the main concepts and definitions as introduced in [CKMM04].

1.1.1 Informal presentation of production systems
A production system consists mainly of the following five components:

• The Fact Types are user defined datatypes, like structs with fields or properties. There are intended for organizing the data that will be manipulated, for instance, we can have a fact type representing a house with properties like color, price, availability, and so on. But, notice that in most cases, we are restricted to basic types for the properties, so it could not be possible to have a property of type address in the house fact type defined before, if the address is a composed data type.

We can then view a fact as a concrete assignment of values to the properties for a given fact type, for instance, an available red house that costs one thousand.

• The Working Memory (WM) is the current program state, it is a global structure of facts. We will see later that this structure could be implemented either by sets or multisets.

• Production Rules are conditional statements of the form

   [Name] if Condition then Action

   A rule has a name and it acts by addition and retraction of facts on the WM iff the condition is fulfilled. Here the condition is usually called the left hand side (LHS) of the production rule and the action its right hand side (RHS). The condition may or not be satisfied by the WM as described in the next section together with more precise explanation for condition and action. When the LHS of a rule is satisfied, the rule is said to be activated.

• The Production Memory (PM) is a structure of production rules, also known as Knowledge Base. It is almost always unvarying, in spite of some production system implementations that provide facilities to manipulate the production memory as RHS actions.

• A Resolution Strategy that consists of an algorithm for selecting just one rule to execute, if the conditions of the LHS of more than one rule are satisfied at the same time.

The production system interpreter executes a production system by performing a sequence of operations called recognize-act cycle or inference cycle.
1. **Matching**: evaluate the LHS of each rule to determine which ones are activated given the current state of the WM. This is the most time consuming step in the execution of a production system, and here is were the rete algorithm is used.

2. **Conflict Resolution or Selection**: select one activated rule. If no rule is activated, halt the interpreter returning the current state of the WM.

3. **Firing or Act**: perform the actions specified in the RHS of the selected rule.

4. go to step 1.

When a rule is activated, an instantiation\(^1\) is generated as an ordered pair of the form:

<rule, list of facts that satisfy its LHS>.

Instantiations are maintained in the Conflict Set (CS). Then, the Resolution Strategy selects just one rule of this set, and its RHS is executed; it is said that the rule is fired.

A schematic view of the data work flow in a production system is shown in Figure 1.

![Figure 1: production system work flow](image)

1.1.2 **Formal Description of Production Systems**

Now that the need for a formal definition of production systems is clear, we will give a formal description of production systems.

We consider a set \( \mathbb{F} \) of function symbols, usually denoted \( f, g, h, \ldots \), a set \( \mathbb{P} \) of predicate symbols, and infinite sets \( X \) and \( \mathcal{L} \) respectively called set of variables and of labels. Variables are denoted \( x, y, z, \ldots \). In most of the practical situations, finite set of labels are enough. These sets are assumed to be disjoint. Each function symbol and predicate symbol has a fixed arity. Nullary function symbols are called constants. We assume that there is at least one constant. The set of terms (denoted \( T(\mathbb{F}, X) \)), ground terms (denoted \( T(\mathbb{F}) \)), atomic propositions, literals (i.e., atomic proposition or their negation), propositions, sentences (i.e., closed propositions) are defined as usual in term rewriting [KK99, BN98, "T"02] and first-order logic [Gal86].

We will freely use the usual notion of substitution. Notice that since in general first order propositions are instantiated, the substitution mechanism works modulo alpha-conversion to take care of the variable bindings.

**Definition 1.1** A fact \( f \) is a ground term, \( f \in T(\mathbb{F}) \).

**Definition 1.2** We call working memory (WM) a set of facts, i.e. it is a subset of the Herbrand universe defined on the signature.

---

\(^1\)this is a historical name, that does not reflect the common meaning of instantiation
Definition 1.3 A production rule or simply rule or production, denoted

\[ [l] \rightarrow p, c \rightarrow r \rightarrow a \]

consists of the following components.

- A name from the label set: \( l \in \mathcal{L} \).
- A set of positive or negative patterns \( p = p^+ \cup p^- \) where a pattern is a term \( p_t \in T(\mathcal{F}, \lambda) \) and a negated pattern is denoted \( \neg p_t \). \( p^+ \) is the set of all negated patterns and \( p^- \) is the set of the remaining patterns.
- A proposition \( c \)
- A set \( r \) of terms whose instances could be intuitively considered as intended to be removed from the working memory when the rule is fired, \( r = \{ r_i \}_{i \in L} \), where \( \text{Var}(r) \subseteq \text{Var}(p) \cup \text{Var}(c) \).
- A set \( a \) of terms whose instances could be intuitively considered as intended to be added to the working memory when the rule is fired, \( a = \{ a_i \}_{i \in L} \), where \( \text{Var}(a) \subseteq \text{Var}(p) \cup \text{Var}(c) \).

Such a rule is also denoted \( [l] p, c \rightarrow r, a \).

Definition 1.4 We call production memory (\( \mathcal{P} \mathcal{M} \)) a set of production rules.

Remark 1.1 Indeed in the previous definitions, one can discuss the choice of set as the data structure to represent the \( \mathcal{W} \mathcal{M}, p, r, a \) and \( \mathcal{P} \mathcal{M} \).

Definition 1.5 Given a set of facts \( S \) and a set of positive patterns \( p^+ \), \( p^+ \) is said to match \( S \) with respect to a theory \( T \) and a substitution \( \sigma \), written \( p^+ \triangleleft_T S \), if:

\[ \forall p \in p^+ \quad \exists t \in S \quad | \sigma(p) = T t | \]

We say that a set of negative patterns \( p^- \) dis-matches a set of facts \( S \), written \( p^- \triangleright_T S \), iff:

\[ \forall \neg p \in p^- \quad \forall t \in S \quad \forall \sigma \quad | \sigma(p) \triangleright_T t | \]

Definition 1.6 Given a substitution \( \sigma \), so that \( \text{Dom}(\sigma) = \text{Var}(p) \cup \text{Var}(c) \), a production rule \( [l] p, c \rightarrow r, a \) is \( (\sigma, \mathcal{W} \mathcal{M}) \)-fireable on a working memory \( \mathcal{W} \mathcal{M} \) when

1. \( p^+ \triangleleft_T \mathcal{W} \mathcal{M} \)
2. \( \mathcal{T} \models \sigma(c) \)
3. \( \sigma p^- \triangleright_T \mathcal{W} \mathcal{M} \)

for a minimal (with respect to the subset ordering) subset \( \mathcal{W} \mathcal{M}' \) of \( \mathcal{W} \mathcal{M} \) and a matching theory, \( \mathcal{T}' \), and an additional theory \( \mathcal{T} \) so that \( \mathcal{T} \subseteq \mathcal{T}' \), for the condition \( c \). A fireable rule is also called an activation.

Definition 1.7 Given a production rule \( [l] p, c \rightarrow r, a \) which is \( (\sigma, \mathcal{W} \mathcal{M}) \)-fireable on a working memory \( \mathcal{W} \mathcal{M} \), its application leads to the new working memory \( \mathcal{W} \mathcal{M}'' \) defined as:

\[ \mathcal{W} \mathcal{M}'' = (\mathcal{W} \mathcal{M} \sigma(r) \cup \sigma(a) \]

This is denoted \( \mathcal{W} \mathcal{M} \circ_{[l] p, c \rightarrow r, a} \mathcal{W} \mathcal{M}'' \) or simply \( \mathcal{W} \mathcal{M} \Rightarrow \mathcal{W} \mathcal{M}'' \). The couple \( (\sigma(r), \sigma(a)) \) is called the \( (\sigma, \mathcal{W} \mathcal{M}) \)-action of the production rule \( [l] p, c \rightarrow r, a \) on the working memory \( \mathcal{W} \mathcal{M} \).

Definition 1.8 For a given working memory \( \mathcal{W} \mathcal{M} \) and a set of production rules \( \mathcal{R} \), the set

\[ CS = \{ (l, \{ f_1, \ldots, f_k \}) \mid \{ [l] p, c \rightarrow a, r \} \in \mathcal{R} \land \exists r \text{ so that } l \text{ is } (\sigma, \mathcal{W} \mathcal{M})\text{-fireable on } \mathcal{W} \mathcal{M} \} \]

is called the \( \mathcal{R} \circ \mathcal{W} \mathcal{M} \)-conflict set, where \( \mathcal{W} \mathcal{M} = \{ f_1, \ldots, f_k \} \)
A conflict set could be either empty (no rule is fireable), unitary (only one rule can fire), finite (a finite number of rule is activated) or infinitary (an infinite number of matches could be found due to the theory modulo which we work [FH86]). Whether finite or infinite, one should decide which rule should be applied: this is one of the major topics of interest in production systems, addressed by resolution strategies.

**Definition 1.9** A resolution strategy is a computable function that given a set of production rules \( \mathcal{R} \) and a production derivation

\[
\mathcal{WM}_0 \Rightarrow \mathcal{WM}_1 \Rightarrow \cdots \Rightarrow \mathcal{WM}_n
\]

returns a unique element of the \( \mathcal{R}@\mathcal{WM}_n \)-conflict set.

We have now all the ingredients to provide a general definition of production systems:

**Definition 1.10** A production system is defined as

\[
\mathcal{GPS} = (\mathcal{P}, \mathcal{F}, \mathcal{X}, \mathcal{L}, \mathcal{WM}_0, \mathcal{PM}, S, \mathcal{T}, \mathcal{T}')
\]

Where:

- \( \mathcal{P} \) is the set of predicate symbols,
- \( \mathcal{F} \) is the set of function symbols,
- \( \mathcal{X} \) is the set of variables,
- \( \mathcal{L} \) is the set of labels,
- \( \mathcal{WM}_0 \) is the initial working memory,
- \( \mathcal{PM} \) is the production memory over \( \mathcal{H} = (\mathcal{F}, \mathcal{P}, \mathcal{X}, \mathcal{L}) \),
- \( \mathcal{S} \) is the resolution strategy,
- \( \mathcal{T} \) is the matching theory,
- \( \mathcal{T}' \) is the constraint theory, and \( \mathcal{T} \subseteq \mathcal{T}' \).

**Remark 1.2** This definition of a production system is quite general as facts and patterns can be deep and non-linear, conditions can be any arbitrary first-order propositions, and resolution strategies can take the full derivation history into account.

**Definition 1.11** The Inference Cycle is described in figure 2.

![Inference Cycle](image)

Figure 2: Inference Cycle

---

Projet RNTL : MANIFICO/Sous Projet 1/Fourniture 1.2/V1.0
ILOG, INRIA Lorraine, INRIA Rocquencourt
1.2 Rewrite Systems

2 Concepts Comparison

2.1 Terms

Same idea in general, despite most implementations of production systems have some limitations handling deeper terms. On other side, production systems have also the so called WM, which could be emulated by rewrite system using a context variable P, and an additional commutative symbol U.

2.2 Patterns

Same idea in general, despite most implementations of production systems have some limitations handling deeper patterns. On other side, production systems have the concept of negative patterns, which is related to natural existence of the WM as a context. This allows us to have patterns that should dismatch the context variable. This behaviour could be emulated by rewrite systems using a conditional rule controlled by a special strategy.

2.3 Conditions

Here we should consider two different rewrite systems:

- General Rewrite Systems
- Conditional Rewrite Systems

The conditions for a general production systems can be any proposition, but, of course, the implementations can have some limitations, in special, with respect to the use of variables not binded by the matching process. Which will need, in general, an external Constraint Solver for handling them.

2.4 Matching

In this aspect, we have two main differences:

- Searching: the most important difference during the matching process is that rewrite systems naturally match subterms. For instance, if we have the following rewrite rule:

\[ f(x) \rightarrow f(x + 2) \]

and the term:

\[ g(f(3)) \]

then, the term will be rewritten into

\[ g(f(5)) \]

and this is not possible in production systems, where the whole term has to match a whole pattern. For a more detailed analysis of this, see section 3.2.

- Dismatching: as described in section 2.2.

2.5 Rules

The first difference between both systems with respect to a rule is that the syntax for production rules already includes explicitly the conditions, which is not the case of rewrite systems, where we can have two different kinds of rules:

- Rewrite rules
• Conditional rewrite rules

Another difference is that production rules have to explicitly indicate any transformation: remove and/or add the terms, where rewrite rules implicitly removes the matched term(s).

2.6 Strategies

With respect to strategies, first there is a theoretical definition for production systems, as any computable function as described in Section 1.9. And then, there are the different implementations of production systems, which use mainly the following criteria:

• An explicitly given priority,
• Rule activation time, and
• Rule complexity

2.7 Theories

what is about matching theories? and theories for handling the conditions?

3 Translation

3.1 Production Systems into Rewrite Systems

3.1.1 Snyder and Schmolze’s approach

We summarize here the approach proposed by Snyder and Schmolze in [SS96].

The three basic components of a Production System:

• The working memory
• The collection of production rules
• The interpreter for applying rules repeatedly to the memory, subject to some conflict resolution strategy.

are represented, using the main notion of associative and commutative terms, in the following way.

**Working Memory** The memory is considered to be a set² of positive ground atoms, completed with negatives according to the Reiter’s Closed World Assumption.

\[ W = P \cup N \]

where:

• \( P \) is a finite set of ground atoms (the actual memory facts), and
• \( N = \{ A | A \notin P \} \) (all other negative facts)

²The Snyder and Schmolze’s full paper also considers the use of a multiset
Production Rules. Depending on the fact that a production rule contains a Negation-As-Failure (NAF) condition\(^3\) or not, it may be translated into two different types of rewrite rules:

- Preserving rewrite rules.
  - For production rules without a NAF condition, has the form:
    \[ L_1, \ldots, L_n \rightarrow L'_1, \ldots, L'_n \]
  - where on each side of the arrow we have sets of literals (possibly containing variables) and where:
    \[ \forall i L'_i = L_i \lor L'_i = L_i \]
  - In other words, the set of atoms is the same on each side, but the sign of a particular atom may have flipped from positive to negative (representing a deletion), from negative to positive (representing an addition), or has remained the same (representing a side condition). The application of such a rewrite rule preserves the property of being a working memory.
  - All production rules that only test for membership, only add or delete facts, and do not have NAF conditions, can be formalized by one or more preserving rewrite rules.\(^4\)

- Constrained preserving rules.
  - This kind of rules have the form \( S \rightarrow T\square \varphi \) where \( S \rightarrow T \) is a preserving rewrite rule and \( \varphi \) is a first-order formula where every free variable in \( \varphi \) occurs in \( S \) (and thus in \( T \)).
  - However, we only need constraints of a particular form; roughly, we will transform each NAF condition \( L_i \) involving NAF variables \( x_{i,1}, \ldots, x_{i,n} \) into a subconstraint of the form \( \varphi_i = \forall x_{i,1}, \ldots, x_{i,n} L_i \) where none of the \( x_{i,j} \) appear in \( S \setminus \{ L_i \} \). This results in a constraint of the form \( \square [\varphi_1 \land \cdots \land \varphi_n] \) for a production rule with \( n \) NAF conditions.
  - But some NAF variable \( x_{i,j} \) may be constrained by some condition \( C \) in \( L_i \) if this is the case, the constraint \( \varphi_j \) should be expanded to \( \varphi_j \lor \neg C \).

Production Rule Interpreter. We formalize the last main feature of a production rule system as a rewriting process subject to checking the constraints.

We say that a WM \( W \) rewrites to \( W' \) with respect to a set of constrained rules \( R \), denoted \( M \rightarrow_R M' \), if

- there exists a rule \( P \rightarrow T\square \varphi \) in \( R \),

- \( M \) has the form \( M[\sigma] \) for some substitution \( \sigma \) such that \( M \models \varphi \sigma \), and

- \( M' \) has the form \( M[T\sigma] \).

The subset \( \sigma \) is called the redex.

Given this, we have a good approximation of a production rule system as an AC-Rewrite system, but we do not model the conflict resolution strategy.

Since we are acting on term rewriting (in this case modulo AC), the resolution of conflicts could be done in at least two ways.

The first one consists of having the production system encoded as a confluent AC-rewrite system. In this case, there is always a common reduct for any two terms issued from a common ancestor. However, the rewrite system is terminating, the result is uniquely defined, independently of the rewrite strategy in use. AC-term rewriting and its meta-properties have been extensively studied from AC-unification [St87], AC-completion [PS81, Hu80] (generalizing Knuth and Bendix approach [KB70]), to AC-termination [RN93, Rub02]. Moreover, AC-matching [Hu79, Eke95] can be efficiently compiled [KM01]. Even if the theoretical background of confluent and terminating rewrite system is now well investigated, this approach has the drawback that, indeed, only few production systems are in essence confluent and terminating.

The second one consists in using explicitly rewrite strategies for explicitly guiding the term rewriting process. This is typical of the approach taken in the ELAN system where rewrite rule

---

\(^3\)A negative condition which contains a variable which appears nowhere else.

\(^4\)More than one rule may be necessary, since a rule might add an atom without testing for its absence.
are split in two categories. The first one called "un-labeled rules" are used for computation and are therefore assumed to form a confluent and terminating rewrite system. The second kind of rules are "labeled rules" and are assumed to be controlled by a strategy using the labels of the rule to fire the appropriate one. This is described in particular in [KKV94, BCD'04] and gave rise to the rewriting calculus [CK01], generalizing term rewriting as well as lambda-calculus and where strategy guided rewriting can be given a precise semantics [CKLW03]. Moreover (symbolic) constraint rewriting could be simply described in this setting, providing an explicit handling of substitutions and exceptions [FK02, CFK04] but also of structure with sharing and cycles [BBCK04].

For a complete example of such an encoding, see the appendix Section 4.8 page 22.

3.1.2 Using an Matching Constraint Approach

We model a production system using an approach simplifying the one by Schwolze and Snyder. The main originality is the introduction of a new predicate "don't match" which is true iff a condition does not match any fact in the working memory. It is therefore a dis-matching problem that shares many similarities with dis-unification one [Com91].

**Definition 3.1** Let \( s, t \) be terms and \( T \) a theory. A matching constraint, denoted \( s \prec_T^? t \), has a solution \( \sigma \) when \( \sigma(s) =_T t \). When a matching constraint is unsatisfiable (i.e., has no solution), this is denoted \( s \nprec_T^? t \): there exist no substitution \( \sigma \) such that \( \sigma(s) =_T t \). When \( T \) is the empty theory, it is just omitted.

A production system can be then modeled is the following way:
Propositions are considered as terms of type `bool`.
The working memory is a term, denoted `WM` and built using the constructors:

- `nil`
- an associative and commutative operator "\( , \)" with neutral element `nil`.

A production rule \([l] \text{ if } p \text{ remove } r \text{ add } a \text{ is } C\) is a conditional rewrite rule
\[
c, r \rightarrow a \text{ if } C
\]
with the standard semantics of AC-rewriting, that is:

\[
WM \rightarrow_{c, r \rightarrow a} \text{ if } C \text{ if } WM' \Leftrightarrow \exists \sigma, \sigma(c) =_C WM \text{ and } WM' = \sigma(a) \text{, provided } \sigma(C)
\]

See Section 4.8.2 on page 23 for a simple example of this encoding.

3.2 Rewrite Systems into Production Systems

Given the Fibonacci example implemented in a rewrite system:

\[
\begin{align*}
\text{fib}(0) & \rightarrow 1 \\
\text{fib}(1) & \rightarrow 1 \\
\text{fib}(n+2) & \rightarrow \text{fib}(n+1) + \text{fib}(n)
\end{align*}
\]

or more accurate:

\[
\begin{align*}
\text{plus}(x, \text{ZERO}) & \rightarrow x \\
\text{plus}(x, \text{suc}(y)) & \rightarrow \text{suc}(\text{plus}(x, y)) \\
\text{fib}(\text{ZERO}) & \rightarrow \text{suc}(\text{ZERO}) \\
\text{fib}(\text{suc}(\text{ZERO})) & \rightarrow \text{suc}(\text{ZERO}) \\
\text{fib}(\text{suc}(\text{suc}(n))) & \rightarrow \text{plus}(\text{fib}(\text{suc}(n)), \text{fib}(n))
\end{align*}
\]

Projet RNTL : MANIFICO / Sous Projet 1 / Fourniture 1.2 / V1.0
ILOG, INRIA Lorraine, INRIA Rocquencourt
where, in this case, $\text{plus}/2$ does not need to be nor assoc nor comm.

How it works step by step in a rewrite system?

For instance, given $\text{fib}(3) = \text{fib}(\text{suc}(\text{suc}(\text{suc}(\text{ZERO}))))$:

<table>
<thead>
<tr>
<th>term</th>
<th>applied rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fib}(\text{suc}(\text{suc}(\text{ZERO})))$</td>
<td></td>
</tr>
<tr>
<td>$\text{plus}(\text{fib}(\text{suc}(\text{suc}(\text{ZERO}))), \text{fib}(\text{suc}(\text{suc}(\text{ZERO}))))$</td>
<td>rule 1e</td>
</tr>
<tr>
<td>$\text{plus}(\text{fib}(\text{suc}(\text{suc}(\text{ZERO}))), \text{suc}(\text{ZERO}))$</td>
<td>rule 1d</td>
</tr>
<tr>
<td>$\text{suc}(\text{plus}(\text{fib}(\text{suc}(\text{suc}(\text{ZERO}))), \text{suc}(\text{ZERO})))$</td>
<td>rule 1b</td>
</tr>
<tr>
<td>$\text{suc}(\text{fib}(\text{suc}(\text{suc}(\text{ZERO}))))$</td>
<td>rule 1a</td>
</tr>
<tr>
<td>$\text{fib}(\text{suc}(\text{suc}(\text{ZERO})))$</td>
<td>rule 1e</td>
</tr>
<tr>
<td>$\text{suc}(\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{suc}(\text{ZERO})))$</td>
<td>rule 1d</td>
</tr>
<tr>
<td>$\text{suc}(\text{fib}(\text{suc}(\text{suc}(\text{ZERO}))))$</td>
<td>rule 1b</td>
</tr>
<tr>
<td>$\text{suc}(\text{suc}(\text{fib}(\text{suc}(\text{suc}(\text{ZERO}))))$</td>
<td>rule 1a</td>
</tr>
<tr>
<td>$\text{suc}(\text{suc}(\text{suc}(\text{ZERO})))$</td>
<td>rule 1c</td>
</tr>
</tbody>
</table>

assuming an standard in-order strategy.

The main problem for trying to do something similar in a production system is that rete is still unable to search inside a term and handle subterm matching, the same for assoc and/or comm symbols.

This means that it is impossible to directly translate a general rewrite system into a production system.

So, there are 2 things to do:

- to propose a restricted rewrite system so that it can be translated into a production system, restricting searching inside of term.

First we can just say that we can not handle searching, or the another possibility is to handle by hand the necessary searching by additional rules. So, if we know that the height of your terms will be restricted by some number $h$, we can write additional production rules for the given combinations. The number of this additional rules will depend on $h$ and also the amount of symbols our system is using and how they interact. In general, it will be a exponential number of additional rules.

For instance, for our Fibonacci example, taking into account all the searching needed for rewrite $\text{fib}(\text{suc}(\text{suc}(\text{suc}(\text{ZERO}))))$, we should have the following rules, including the original 5 rules and adding some additional ones for searching:

\[
\begin{align*}
\text{plus}(x, \text{ZERO}) & \quad \Rightarrow \quad x & \text{(2a)}
\text{plus}(x, \text{suc}(y)) & \quad \Rightarrow \quad \text{suc}(\text{plus}(x, y)) & \text{(2b)}
\text{fib}(\text{ZERO}) & \quad \Rightarrow \quad \text{suc}(\text{ZERO}) & \text{(2c)}
\text{fib}(\text{suc}(\text{ZERO})) & \quad \Rightarrow \quad \text{suc}(\text{ZERO}) & \text{(2d)}
\text{fib}(\text{suc}(\text{suc}(n))) & \quad \Rightarrow \quad \text{plus}(\text{fib}(\text{suc}(n)), \text{fib}(n)) & \text{(2e)}
\text{plus}(x, \text{fib}(\text{suc}(\text{ZERO}))) & \quad \Rightarrow \quad \text{plus}(x, \text{suc}(\text{ZERO})) & \text{(2f)}
\text{suc}(\text{plus}(x, \text{ZERO})) & \quad \Rightarrow \quad \text{plus}(x) & \text{(2g)}
\text{suc}(\text{plus}(x, \text{suc}(n))) & \quad \Rightarrow \quad \text{suc}(\text{plus}(\text{fib}(\text{suc}(n)), \text{fib}(n))) & \text{(2h)}
\text{suc}(\text{plus}(x, \text{fib}(\text{ZERO}))) & \quad \Rightarrow \quad \text{suc}(\text{plus}(x, \text{suc}(\text{ZERO}))) & \text{(2i)}
\text{suc}(\text{plus}(x, \text{suc}(y))) & \quad \Rightarrow \quad \text{suc}(\text{plus}(x, y)) & \text{(2j)}
\text{suc}(\text{suc}(\text{plus}(x, \text{ZERO}))) & \quad \Rightarrow \quad \text{suc}(\text{suc}(\text{plus}(x))) & \text{(2k)}
\text{suc}(\text{suc}(\text{fib}(\text{ZERO}))) & \quad \Rightarrow \quad \text{suc}(\text{suc}(\text{fib}(\text{ZERO}))) & \text{(2l)}
\end{align*}
\]

Where, $\Rightarrow$ means production rule, and additionally means that the term on the LHS will be removed from the WM and that the term on the RHS will be added if a term matches the LHS.
and the execution would be:

<table>
<thead>
<tr>
<th>term</th>
<th>applied rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fib} (\text{suc}(\text{suc}(\text{ZERO})))$</td>
<td></td>
</tr>
<tr>
<td>$\text{plus} (\text{fib}(\text{suc}(\text{ZERO})), \text{fib}(\text{suc}(\text{ZERO})))$</td>
<td>rule 2e</td>
</tr>
<tr>
<td>$\text{plus} (\text{fib}(\text{suc}(\text{ZERO})), \text{suc}(\text{ZERO}))$</td>
<td>rule 2f</td>
</tr>
<tr>
<td>$\text{suc}(\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{ZERO}))$</td>
<td>rule 2b</td>
</tr>
<tr>
<td>$\text{suc}(\text{fib}(\text{suc}(\text{ZERO})))$</td>
<td>rule 2g</td>
</tr>
<tr>
<td>$\text{suc}(\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{fib}(\text{ZERO})))$</td>
<td>rule 2h</td>
</tr>
<tr>
<td>$\text{suc}(\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{suc}(\text{ZERO})))$</td>
<td>rule 2i</td>
</tr>
<tr>
<td>$\text{suc}(\text{fib}(\text{suc}(\text{ZERO})))$</td>
<td>rule 2k</td>
</tr>
<tr>
<td>$\text{suc}(\text{suc}(\text{fib}(\text{suc}(\text{ZERO}))))$</td>
<td>rule 2l</td>
</tr>
</tbody>
</table>

But the given set of rules is still very limiting. For instance, $\text{fib}(2) = \text{fib}(\text{suc}(\text{suc}(\text{ZERO})))$, can not be handled:

<table>
<thead>
<tr>
<th>term</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fib}(\text{suc}(\text{ZERO})))$</td>
<td>rule 2e</td>
</tr>
<tr>
<td>$\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{fib}(\text{ZERO})))$</td>
<td></td>
</tr>
</tbody>
</table>

And there is no rule for rewriting that term.

So, to being able to rewrite $\text{fib}(2) = \text{fib}(\text{suc}(\text{suc}(\text{ZERO})))$ we should write the following additional rules:

$$\text{plus}(x, \text{fib}(\text{ZERO})) \implies \text{plus}(x, \text{suc}(\text{ZERO})) \quad (3a)$$
$$\text{suc}(\text{fib}(\text{suc}(\text{ZERO}))) \implies \text{suc}(\text{suc}(\text{ZERO})) \quad (3b)$$

And the execution would be:

<table>
<thead>
<tr>
<th>term</th>
<th>applied rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fib}(\text{suc}(\text{ZERO})))$</td>
<td></td>
</tr>
<tr>
<td>$\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{fib}(\text{ZERO})))$</td>
<td>rule 2e</td>
</tr>
<tr>
<td>$\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{suc}(\text{ZERO}))$</td>
<td>rule 3a</td>
</tr>
<tr>
<td>$\text{suc}(\text{plus}(\text{fib}(\text{suc}(\text{ZERO})), \text{ZERO}))$</td>
<td>rule 2b</td>
</tr>
<tr>
<td>$\text{suc}(\text{fib}(\text{suc}(\text{ZERO})))$</td>
<td>rule 2g</td>
</tr>
<tr>
<td>$\text{suc}(\text{suc}(\text{fib}(\text{suc}(\text{ZERO}))))$</td>
<td>rule 3b</td>
</tr>
</tbody>
</table>

- to propose improvements to the rete algorithm to be able to do the same as a general rewrite system.

The first idea for extending rete for searching is to decompose recursively the terms in the $\mathcal{WM}$ and to have an auxiliary $\mathcal{WM}$ where the decomposed subterms are intenened to match, and then to merge the results in and coherent way.

Some interesting aspects of this approach are:

- what to put in the auxiliary $\mathcal{WM}$? just the decomposed subterms? or also the content of the main $\mathcal{WM}$?
- will we need only one auxiliary $\mathcal{WM}$ or more, one for each decomposed subterm? more over, will we need to keep independent memories of the rete execution in the auxiliary $\mathcal{WM}$ or not, from one firing to another??
- what is about $\mathcal{PM}$? should we use the same $\mathcal{PM}$ for rewriting subterms? or should we may be have also one or more auxiliary $\mathcal{PM}$s?
- and also performance and memory usage in comparison to rewrite systems.
4 Examples

This appendix shows a same toy example, first in abstract notation and then implemented in several production system.

4.1 Abstract Language

\[ F := \{
    \text{house}/4, \text{houseaddress}/4, \text{myaddress}/3, \text{war}/2, \text{searching}/0, \\
    \text{red}/0, \text{blue}/0, \text{usa}/0, \text{iran}/0, \text{france}/0
\} \]

\[ W_{M_0} := \{
    \text{house}(1, \text{red}, 341, \text{true}), \text{houseaddress}(1, 251, "rue jeanne d'arc", "nancy"), \\
    \text{house}(2, \text{blue}, 390, \text{true}), \text{houseaddress}(2, 121, "avenue de brabois", "villers les nancy"), \\
    \text{house}(3, \text{red}, 415, \text{true}), \text{houseaddress}(3, 31, "rue cannot", "condroz les nancy"), \\
    \text{myaddress}(2551, "gorbea", "santiago"), \\
    \text{war}(\text{usa}, \text{iran}), \\
    \text{searching}()
\} \]

\[ PM := \{
    \text{[HouseSearch]}
    \begin{align*}
    \text{searching}() \land & \\
    \text{house}(\text{id}, \text{red}, \text{price}, \text{true}) \land & \\
    \text{houseaddress}(\text{id}, \text{number}, \text{street}, \text{city}) \land & \\
    \text{myaddress}(\text{nn}, \text{ns}, \text{nc}) \land & \\
    \neg\text{war}(\text{s}1, \text{france}) \land & \\
    \neg\text{war}(\text{france}, \text{s}2), & \\
    \text{price} < 400 & \implies
    \end{align*}
\}

\[
\begin{align*}
    \{ & \\
    \text{searching}(), & \\
    \text{house}(\text{id}, \text{red}, \text{price}, \text{true}), & \\
    \text{myaddress}(\text{nn}, \text{ns}, \text{nc}) & \\
    \}, & \\
    \{ & \\
    \text{house}(\text{id}, \text{red}, \text{price}, \text{false}), & \\
    \text{myaddress}(\text{number}, \text{street}, \text{city}) & \\
    \}
\end{align*}
\]

4.2 OPS5

OPS5 Core Syntax Explanations

- ; : line comment.
- (literalize <factname> <slotname>*): used for defining a new type of fact called <factname>, with 0, 1 or more slots (or properties) <slotname>*.
- (p <ruleName>\? <pattern>* \rightarrow <action>*): used for defining a rule called <ruleName>, with conditions or patterns <pattern>* and actions <action>*.

- (make <factName> ['<slotName> <value>]*): used for creating a new fact, called <factName>, adding it to the WM.

- (remove <cid>): used for retracting the fact that satisfy the condition given by <cid>, which is just the number of the condition in the LHS.

- (modify <cid> ['<slotName> <newValue>*]): used for modifying the fact that satisfy the condition given by <cid>. Modifying just the values of the given slots by `<slotName>` replacing the value with `<newValue>`. Internally implemented as a (remove <cid>) followed by a (make ...).

**OPS5 Example**

```plaintext
; new type(fact) definitions
; a fact called house( representing a house ) with 4 "properties": id, color, price and forrent
(literalize house id color price forrent)
; a fact called houseaddress( representing the address of a house ) with 4 "properties":
; id, number, street and city
(literalize houseaddress id number street city)
; a fact called myaddress( representing my address ) with 3 "properties": number, street and city
(literalize myaddress number street city)
; a fact called searching with no properties, just for indicating that i am looking for a new house
(literalize searching)
; a fact called war( representing a war between 2 sides ) with 2 "properties": sidel and side2
(literalize war sidel side2)

; a rule with the name search_for_house
(p search_for_house
  (if the searching flag is set
    (searching)
    ; search for a red house with a price less than 400
    (house id <id> color red price < 400 forrent true)
    ; get the address of that house
    (houseaddress id <id> number <num> street <str> city <cit>)
    ; get my address
    (myaddress)
    ; if there is no war where france is involved
    (war "sidel france"
     (war "side2 france"
     (->)
     ; i found a nice house so i delete the searching fact
     (remove 1)
     ; modify the state of the house
     (modify 2 "forrent false")
     ; modify my address
     (modify 4 "number <num> street <str> city <cit>

; working memory initialization
; first house
(make house id 1 color red price 341 forrent true)
(make houseaddress id 1 number 251 street "Rue Jeanne D'Arc" city Nancy)
; second house
(make house id 2 color blue price 399 forrent true)
(make houseaddress id 2 number 121 street "Avenue de Brabois" city Villers-les-Nancy)
; third house
(make house id 3 color red price 415 forrent true)
(make houseaddress id 3 number 31 street "Rue Cantor" city Vandoeuvre-les-Nancy)
; my address
(make myaddress number 2551 street Gorbeia city Santiago)
; searching flag
(make searching)
; war
(make war "sidel usa" "side2 irak")
```
4.3 CLIPS

CLIPS Core Syntax Explanations

- ;: line comment.
- (deftemplate <factname> (slot <slotname>)*): used for defining a new type of fact
called <factname>, with 0, 1 or more slots (or properties) <slotname>.
- (defrule <rulename>?! ([<id> <- ]<pattern>* => <action>*): used for defining a
rule called <rulename>, with conditions or patterns <pattern> which can be associated to
a given condition id <id>, and actions <action>*.
- (assert <factname> (<slotname> <value>*)): used for creating a new fact, called <factname>,
adding it to the WM.
- (deffacts <factgroupname> (<factname> (<slotname> <value>*))): used for cre-
ting several facts at once.
- (retract <id>): used for retracting the fact that satisfy the condition given by <id>.
- (modify <id> (<slotname> <newvalue)*): used for modifying the fact that satisfy
the condition given by <id>. Modifying just the values of the given slots by <slotname> replac-
ing the value with <newvalue>. Internally implemented as a (retract <id>) followed by
a (make ...).

CLIPS Example

;new type(fact) definitions
;a fact called house representing a house) with 4 "properties": id, color, price and forrent
(deftemplate house (slot id) (slot color) (slot price) (slot forrent))
;a fact called houseaddress representing the address of a house) with 4 "properties":
id, number, street and city
(deftemplate houseaddress (slot id) (slot number) (slot street) (slot city))
;a fact called myaddress representing my address) with 3 "properties": number, street and city
(deftemplate myaddress (slot number) (slot street) (slot city))
;a fact called searching with no properties, just for indicating that i am looking for a new house!
(deftemplate searching)
;a fact called war representing a war between 2 sides) with 2 "properties": sid1 and sid2
(deftemplate war (slot sid1) (slot sid2))

;a rule with the name search_for_house
(defrule search_for_house
 ;if the searching flag is set, and bind the matching fact to ?f1
 (?f1 <- (searching))
 ;search for a red house with a price less than 400, and bind the matching fact to ?f2
 (?f2 <- (house (id ?id) (color red) (price ?p) (forrent true))
  (test (< ?p 400)))
 ;get the address of that house
 (houseaddress (id ?id) (number ?num) (street ?str) (city ?cit))
 ;get my address, and bind the matching fact to ?f3
 (?f3 <- (myaddress))
 ;if there is no war where france is involved
 (not (war (sidel france)))
 (not (war (sid2 france)))
 =>
 ;i found a nice house so i delete the searching fact
 (retract ?f1)
 ;modify the state of the house
 (modify ?f2 (forrent false))
 ;modify my address
 (modify ?f3 (number ?num) (street ?str) (city ?cit))
)

;working memory initialization
(deffacts startup
;first house
(house (id 1) (color red) (price 341) (for rent true))
(houseaddress (id 1) (number 251) (street "Rue Jeanne D'Arc") (city Nancy))
; second house
(house (id 2) (color blue) (price 390) (for rent true))
(houseaddress (id 2) (number 121) (street "Avenue de Brabois") (city Villers-les-Nancy))
; third house
(house (id 3) (color red) (price 415) (for rent true))
(houseaddress (id 3) (number 31) (street "Rue Camot") (city Vandoeuvre-les-Nancy))

my address
(myaddress (number 2551) (street Gorbec) (city Santiago))
; searching flag
(searching)
; war
(war (side1 usa) (side2 irak))

4.4 JeOPS

JeOPS Core Syntax Explanations

- First you have to define one Java class for each type of Fact. They have not to implement or extend nothing special. And you can use any method for accessing their fields, i.e., public attributes, getters and setters, or whatever.

- Then the rules definitions are described in a special class which has to be compiled by jeops, with extension .rules. This class should define all the rules, and each rule should have three parts:

  - declarations: Here you have to specify the type of the involved facts, giving a variable name to each one. This step already does matching. If there are no facts in the WM of a required fact type, the rule is not executed.

  - conditions: In this section you can specify several Java conditions, using the variables defined in the previous section. There is also a section called preconditions which can be additionally used.

  - actions: In this section you can execute actions using the following methods: retract(FactObject), assert(FactObject) and modified(FactObject), the last one should be explicitly called after you modify any relevant information of the given FactObject, calling indirectly the rete algorithm.

- And last, you have to define the main class which should instantiate an instance of the knowledge base generated by JeOPS, and initialize the WM by creating and asserting some Facts, for finally calling the run() method for execution.

JeOPS Example

// definition of Fact classes
public class House {
    public int id; public String color;
    public double price; public boolean forRent;
}

public class HouseAddress {
    public int id, number;
    public String street, city;
}

public class MyAddress {
    public int number;
    public String street, city;
    public String toString() {
        return ""+number", "+street", "+city;"
    }
}

public class Searching {}

public class War {
public String side1, side2;
}

// the rules knowledge base
public ruleBase HouseBase {
  rule SearchForHouse {
    declarations
      searching searching;
      House house;
      HouseAddress houseAddress;
      MyAddress myAddress;
      War war;
    conditions
      house.forRent;
      house.color.equals("red");
      house.price<400.0;
      house.id == houseAddress.id;
      war.side1.equals("France");
      war.side2.equals("France");
    actions
      retract(searching);
      house.forRent=false;
      modified(house);
      myAddress.number=houseAddress.number;
      myAddress.street=houseAddress.street;
      myAddress.city=houseAddress.city;
      modified(myAddress);
  }
}

// main program, initializing the KB and running
public class TestHouses {
  public static void main(String[] args) {
    HouseBase kb = new HouseBase();

    House h1 = new House();
    h1.id=1;h1.color="red";h1.price=341.00;h1.forRent=true;
    HouseAddress ha1 = new HouseAddress();
    ha1.id=1;ha1.number=251;ha1.street="Jeanne D'Arc";ha1.city="Nancy";
    kb.assert(h1);
    kb.assert(ha1);

    House h2 = new House();
    h2.id=2;h2.color="blue";h2.price=356.00;h2.forRent=true;
    HouseAddress ha2 = new HouseAddress();
    ha2.id=2;ha2.number=121;ha2.street="Avenue de Brabois";ha2.city="Villers-les-Nancy";
    kb.assert(h2);
    kb.assert(ha2);

    House h3 = new House();
    h3.id=3;h3.color="red";h3.price=415.00;h3.forRent=true;
    HouseAddress ha3 = new HouseAddress();
    ha3.id=3;ha3.number=31;ha3.street="Rue Carnot";ha3.city="Vandoeuvre-les-Nancy";
    kb.assert(h3);
    kb.assert(ha3);

    MyAddress ma = new MyAddress();
    ma.number=2551;ma.street="Garbon";ma.city="Santiago";
    kb.assert(ma);

    Searching s = new Searching();
    kb.assert(s);

    War w = new War();
    w.side1="USA";w.side2="Iрак";
    kb.assert(w);

    System.out.println("i lived here:\n" + ma);
    kb.run();
    System.out.println("and moved to:\n" + ma);
  }
}
4.5 JRules, using TRL

Using the same fact classes defined in the JeOPS example.

Working Memory Initialization

```java
assert [ ] [ ] House [ ]
  so that id = 1
  and color = "red"
  and price = 341
  and forRent = true

assert [ ] [ ] HouseAddress [ ]
  so that id = 1
  and number = 251
  and street = "Rue Jeanne D Arc"
  and city = "Nancy"

assert [ ] [ ] House [ ]
  so that id = 2
  and color = "blue"
  and price = 390
  and forRent = true

assert [ ] [ ] HouseAddress [ ]
  so that id = 2
  and number = 121
  and street = "Avenue de Brabois"
  and city = "Villers-les-Nancy"

assert [ ] [ ] House [ ]
  so that id = 3
  and color = "red"
  and price = 415
  and forRent = true

assert [ ] [ ] HouseAddress [ ]
  so that id = 3
  and number = 31
  and street = "Rue Carnot"
  and city = "Vandoeuvre-les-Nancy"

assert [ ] [ ] MyAddress [ ]
  so that number = 2551
  and street = "Corbero"
  and city = "Santiago"

assert [ ] [ ] Searching [ ]
  [such that]

assert [ ] [ ] War [ ]
  so that side1 = "USA"
  and side2 = "Iraq"
```

Knowledge Base

```
WHEN
  there is a [ ] Searching [ ] [ called ?f1 ]
    [where]
    [such that]
  there is a [ ] House [ ] [ called ?f2 ]
    where id is called ?id
    such that color = red
    and price < 400
    and forRent = true
  there is a [ ] HouseAddress [ ] [ called ?f3 ]
    [where]
    such that id = ?id
  there is a [ ] MyAddress [ ] [ called ?f4 ]
    [where]
    [such that]
  there is no [ ] War [ ]
    [where]
```
such that side1 = france
       or side2 = france

THEN
   retract ?f1
   modify [ ] ?f2
   so that forRest = false
   modify [ ] ?f4
   so that number = ?f3.number
       and street = ?f3.street
       and city = ?f3.city

ELSE

4.6 ELAN

module houses

import global bool builtinInt;
end

sort Object Space
    House HouseAddress MyAddress Searching War;
end

operators global
    @ U @ : (Space Space) Space (AC);
    empty : Space;
    @ : (Object) Space;

House[h_id=0, h_color=a, h_price=a, h_forRent=a]
    : (builtinInt builtinInt builtinInt builtinInt) House;
    @ : (House) Object;

HouseAddress[ha_id=0, ha_number=a, ha_street=a, ha_city=a]
    : (builtinInt builtinInt builtinInt builtinInt) HouseAddress;
    @ : (HouseAddress) Object;

MyAddress[ma_number=a, ma_street=a, ma_city=a]
    : (builtinInt builtinInt builtinInt) MyAddress;
    @ : (MyAddress) Object;

Searching[]
    @ : Searching;
    @ : (Searching) Object;

War[w_side1=0, w_side2=0]
    : (builtinInt builtinInt) War;
    @ : (War) Object;

go : builtinInt;
result(@) : (Space) builtinInt;
occurWar1(@, @) : (Space builtinInt) bool;
occurWar2(@, @) : (Space builtinInt) bool;
end

strategy global
    loop : <Space> bs;
end

rules for Space
S : space;
id1, id2, pr : builtinInt;
num, str, cit : builtinInt;
num2, str2, cit2 : builtinInt;
global
    [housesearch]
S U House[h_id=id1, h_color=1, h_price=pr, h_forRent=1]
S U HouseAddress[ha_id=ha_id2, ha_number=ha_num, ha_street=ha_street, ha_city=ha_city]
S U MyAddress[ma_number=ma_num2, ma_street=ma_street2, ma_city=ma_city2]
S U Searching[]
=>
S U House[h_id=id1, h_color=1, h_price=pr, h_forRent=0]
U HouseAddress[ha_id=1, ha_number=1, ha_street=str, ha_city=city]
U MyAddress[ma_number=0, ma_street=str, ma_city=city]

if id1 == id2
if pr < 400
if not(occursWar1(S,1))
if not(occursWar2(S,1))
end

rules for builtinInt
S : Space;
num, str, cit : builtinInt;
global
[] go => result(S)
where S := (loop) empty
  U House[h_id=1, h_color=1, h_price=341, h_longitude=1]
  U HouseAddress[ha_id=1, ha_number=251, ha_street=1, ha_city=1]
  U House[h_id=2, h_color=2, h_price=396, h_longitude=1]
  U HouseAddress[ha_id=2, ha_number=21, ha_street=2, ha_city=2]
  U House[h_id=3, h_color=1, h_price=415, h_longitude=1]
  U HouseAddress[ha_id=3, ha_number=31, ha_street=3, ha_city=3]
  U MyAddress[ma_number=2551, ma_street=4, ma_city=4]
  U Searching()
  U War[w_sid=2, w_sid2=3]
end

[] result(S U MyAddress[ma_number=0, ma_street=str, ma_city=cit]) => num end

rules for bool
S : Space;
n, v : builtinInt;
global
[] occursWar1(S U War[w_sid=1, w_sid2=v], n) => true end
[] occursWar1(S, n) => false end
[] occursWar2(S U War[w_sid=1, w_sid2=0], n) => true end
[] occursWar2(S, n) => false end
end

strategies for Space
implicit
[] loop => repeat*(first one(housesearch)) end
end

4.7 JTom

package prodrule;

import atom.*;
import atompure.*;
import prodrule.fib3.fib.*;
import prodrule.fib3.fib.types.*;
import java.util.*;

public class Fib3 {
  private fibFactory factory;

  %Was {
    // extension of adt syntax
    module fib
      imports
        public
          sorts Element
        abstract syntax
            Undef -> Element
  }
Nat(value:Int) -> Element

Fib(arg:Int, val:Element) -> Element


✂️array Space\
  implement { ArrayList }
  getFunSym(t) { (t instanceof ArrayList)?factory.getPureFactory().makeFun(“concElement", 1, false):null }
  mapFunSym(t1,t2) { t1 = t2 }
  equals(l1,l2) { l1.equals(l2) }
  get_element(l,n) { l.get(n) }
  get_size(l) { l.size() }

✂️array Space concElement( Element:* ) { 
  fSym { factory.getPureFactoryO().makeFun("concElement", 1, false) }
  makeEmpty(n) { myEmpty(n) }
  make Append(e,l) { myAdd(e,(ArrayList)l) }
}

private ArrayList myAdd(Object e,ArrayList l) {
  l.add(e);
  return l;
}

private ArrayList myEmpty(int n) {
  ArrayList res = new ArrayList(n);
  return res;
}

public Fib3(fibFactory factory) {
  this.factory = factory;
}

public fibFactory getFibFactory() {
  return factory;
}

public int run(int n) {
  long startChrono = System.currentTimeMillis();
  System.out.println(“running...”);
  ArrayList space = “concElement(Fib(0,Nat(1)), Fib(1,Nat(1)), Fib(0,Undef))”;
  space = loop(space);
  System.out.println("fib" + n + ") = “ + result(space,n) + ” (in " + (System.currentTimeMillis()-startChrono) + " ms)");
  return result(space,n);
}

public ArrayList loop(ArrayList s) {
  ArrayList oldSpace = s;
  ArrayList space = compute(rec(oldSpace));
  while(space != oldSpace) {
    oldSpace = space;
    space = compute(rec(space));
  }
  return space;
}

public final static void main(String[] args) {
  Fib3 test = new Fib3(fibFactory.getInstance(new PureFactory(16)));
  try {
    test.run(Integer.parseInt(args[0]));
  } catch (Exception e) {
    System.out.println("Usage: java Fib <nb>");
    return;
  }
}

public ArrayList rec(ArrayList s) {
  \%match(Space) { 
  concElement(S1#, Fib[arg=n,val=Undef()]), S2#) -> { 
    if( (n>2 && !occursFib(S1#,n-1) && !occursFib(S2#,n-1)) ) {

Projet RNTL : MANHCO / Sous Projet 1 / Fourniture 1.2 / V1.0
ILOG, INRIA Lorraine, INRIA Rocquencourt
4.8 Ac-Rewriting

4.8.1 Snyder and Schmolze’s approach

P := \{ false/0, true/0, not/1, </2, =/2, /2/2 \}
F := \{ house/4, houseaddress/4, myaddress/3, war/2, searching/0 \}

W_P := \{ house(1, red,341, true), houseaddress(1, 251, "Rue Jeanne D’Arc", Nancy),
        house(2, blue,390, true), houseaddress(2, 121, "Avenue de Brabois", Villers-lès-Nancy),
        house(3, red,415, true), houseaddress(3, 31, "Rue Carnot", Vándoeuvre-lès-Nancy),
        myaddress(2551, Corbea, Santiago), searching(), war(usa, irak) \}

W_0 := W_P \cup \{ A | A \in W_P \}
4.8.2 Matching Constraint Approach

We follow here the methodology and encoding presented in section 3.1.2 on page 10.

\[ \mathcal{F} := \{ false/0, true/0, not/1, < /2, = /2/2 \} \cup \{ house/4, houseaddress/4, myaddress/3, war/2, C, searching/0 \} \]

\[ W_0 := \{ \]
\[ house(1, red, 341, true), houseaddress(1, 251, "Rue Jeanne D'Arc", Nancy), \]
\[ house(2, blue, 390, true), houseaddress(2, 121, "Avenue de Brabois", Villers-les-Nancy), \]
\[ house(3, red, 415, true), houseaddress(3, 31, "Rue Carnot", Vandoeuvre-les-Nancy), \]
\[ myaddress(2551, Gorbea, Santiago), searching(), war(usa, irak) \]
\[ \} \]

R := \{ 
  ?P U searching()
  U house(?id, red, ?pr, true)
  U houseaddress(?id, ?num, ?str, ?cit)
  U myaddress(?num, ?ms, ?mc)

  \rightarrow
  ?P
  U house(?id, red, ?pr, false)
  U houseaddress(?id, ?num, ?str, ?cit)
  U myaddress(?num, ?str, ?cit)
  if ?pr < 400 \land war(?s1, france) \neq ?P
\}

5 Benchmark

In this section we benchmark several languages for production rule systems. For this, we use the fibonacci numbers sequence, encoded as following (independent of the resolution strategy):
\[ \mathcal{F} := \{ \text{fib}(2, -2) \} \]
\[ \mathcal{W} \mathcal{M}_0 := \{ \text{fib}(0, 1), \text{fib}(1, 1), \text{fib}(200, -1) \} \]
\[ \mathcal{P} \mathcal{M} := \{ \]

\[ [\text{GoDown}] \]
\[ \text{fib}(n, -1) \wedge \neg \text{fib}(m1, \nu), \]
\[ \nu n = \nu n - 1 \]
\[ \implies \]
\[ \emptyset, \]
\[ \{ \text{fib}(m1, -1) \} \]

\[ [\text{GoUp}] \]
\[ \text{fib}(n, -1) \wedge \text{fib}(m1, \nu1) \wedge \text{fib}(m2, \nu2), \]
\[ \nu n = \nu n - 1 \wedge \nu n > 0 \wedge \nu m = \nu m - 2 \wedge \nu m > 0 \wedge \nu n = \nu n + \nu m \]
\[ \implies \]
\[ \{ \text{fib}(m, -1), \text{fib}(m2, \nu2) \}, \]
\[ \{ \text{fib}(m, \nu) \} \]

The benchmark has been executed on a Pentium IV 1.70Ghz 256Kb L2-cache and 384 MB RAM running Mandrake 9.2, see tables 1 to 4:

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>40 ms</td>
<td>398 pr</td>
<td>9925 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>215 ms</td>
<td>398 pr</td>
<td>1847 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>686 ms</td>
<td>398 pr</td>
<td>580 pr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>810 ms</td>
<td>598 rwr</td>
<td>738 rwr/sec</td>
</tr>
<tr>
<td>JTom 2.0rc2</td>
<td>311 ms</td>
<td>398 pr</td>
<td>1276 pr/sec</td>
</tr>
</tbody>
</table>

Table 1: Fibonacci(200)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>160 ms</td>
<td>798 pr</td>
<td>4988 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>482 ms</td>
<td>798 pr</td>
<td>1654 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>1224 ms</td>
<td>798 rwr</td>
<td>651 rwr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>6590 ms</td>
<td>1197 rwr</td>
<td>181 rwr/sec</td>
</tr>
<tr>
<td>JTom 2.0rc2</td>
<td>1457 ms</td>
<td>798 pr</td>
<td>547 pr/sec</td>
</tr>
</tbody>
</table>

Table 2: Fibonacci(400)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>1010 ms</td>
<td>1997 pr</td>
<td>1977 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>903 ms</td>
<td>1997 pr</td>
<td>2211 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>4332 ms</td>
<td>1997 rwr</td>
<td>460 rwr/sec</td>
</tr>
<tr>
<td>JTom 2.0rc2</td>
<td>18557 ms</td>
<td>1997 pr</td>
<td>107 pr/sec</td>
</tr>
</tbody>
</table>

Table 3: Fibonacci(1000)
5.1 A more demanding benchmark

The same fibonacci sequence, but without garbage collection.

\[ \mathcal{F} := \{ \text{fib}/2, -/2 \} \]
\[ \mathcal{W}_{\mathcal{W}_0} := \{ \text{fib}(0, 1), \text{fib}(1, 1), \text{fib}(200, -1) \} \]
\[ \mathcal{P}_{\mathcal{M}} := \{ \]

\[ \text{[GoDown]} \]
\[ \text{fib}(?n, -1) \land \neg \text{fib}(?m1, ?v), \]
\[ ?n1 = ?n - 1 \]
\[ \Rightarrow \]
\[ \emptyset, \]
\[ \{ \text{fib}(?m1, -1) \} \]

\[ \text{[GoUp]} \]
\[ \text{fib}(?n, -1) \land \text{fib}(?m1, ?v1) \land \text{fib}(?m2, ?v2), \]
\[ ?n1 = ?n - 1 \land ?v1 > 0 \land ?n2 = ?n - 2 \land ?v2 > 0 \land ?v = ?v1 + ?v2 \]
\[ \Rightarrow \]
\[ \{ \text{fib}(?n, -1) \}, \]
\[ \{ \text{fib}(?n, ?v) \} \]

The results are:

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>10 ms</td>
<td>197 pr</td>
<td>19.700 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>244 ms</td>
<td>197 pr</td>
<td>14.28 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>966 ms</td>
<td>1979 pr</td>
<td>20.4 pr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>83780 ms</td>
<td>287 rwr</td>
<td>3 rwr/sec</td>
</tr>
</tbody>
</table>

Table 5: Fibonacci(100)

<table>
<thead>
<tr>
<th>Language</th>
<th>Time [ms]</th>
<th>Rules fired</th>
<th>Rules/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clips 6.1</td>
<td>60 ms</td>
<td>397 pr</td>
<td>66.17 pr/sec</td>
</tr>
<tr>
<td>JeOPS 2.1</td>
<td>370 ms</td>
<td>397 pr</td>
<td>15.88 pr/sec</td>
</tr>
<tr>
<td>JRules 6.0</td>
<td>1171 ms</td>
<td>3971 pr</td>
<td>33.9 pr/sec</td>
</tr>
<tr>
<td>ELAN 3.6g</td>
<td>2454680 ms</td>
<td>597 rwr</td>
<td>0.24 rwr/sec</td>
</tr>
</tbody>
</table>

Table 6: Fibonacci(200)

Acknowledgments: Thanks to François Charpillet for sharing with us his X-tra experience and to the Manifico project members for interactions and comments.
References


\textsuperscript{11}http://www.brics.dk/Projects/CoFI/
\textsuperscript{12}ftp://ftp.brics.dk/Projects/CoFI/


