Operational semantics and compilation of production systems — Fages and Lissajoux's 1992 paper on the Rete algorithm

Francois Fages and Rémi Lissajoux
translated by YKY (Yan King Yin)
general.intelligence@gmail.com

February 16, 2019

Abstract

We present an operational semantics, in the formalism of inferences rules a la Plotkin, for production systems like OPS5 [For81]. We show how these programming languages can be compiled from their operational semantics. We describe how sequential and parallel versions of the Rete algorithm [For82] are formulated and their implementations proved. We show that simple modifications of the formal system described we obtain the Treat algorithm [Mir90] and other optimisations of Rete.

Keywords: production systems, compilation, operational semantics, rule based languages

1 Introduction

The development of expert systems has contributed to the emergence of the pure concept declarative programming software, which can be summarized by Robert Kowalski:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

Rule programming is the best example of this programming style. Unlike a program written in a procedural or functional language in which the sequence of instructions is explicit, a single rule base the logical part of the processing, the sequencing of the instructions is defined separately and is based on general techniques for finding a solution. The tools generating expert systems are languages (or overlanguages) of pro- declarative grammar. The use of these tools makes it possible to develop effectively applications and benefit from powerful compilation techniques. The fact that these techniques ensure excellent performance on standard processors is a decisive point for their integration into conventional computer systems, The relative failure of dedicated machine projects or massively parallel machines for the expert systems shows that the problem is of a purely software.

In fact, rule-based languages now make it possible to reach (in terms of execution time) allowing the use of these languages in "real-time" applications (see [FL91]). These results are due as much to The technological evolution of equipment that has improved the techniques of lation. The performance report between the original version of Ops5 (developed Carnegie Melion University, OpS5 is the ancestor of a number of rule-based languages existing to date [For81)) and the latest version completed [GFK 88] is of two orders of magnitude (& equal machine) [ML91].

To develop and master these compilation techniques more and more complex, it is necessary to have a good understanding of these languages. That's what allows to achieve a semantic study of language, be it semantics which formalizes the execution of a program, or logical semantics which characterizes in an abstract way the result of the execution.

Our effort tends to give a complete semantic analysis of the production systems (of the OPS5 family), and more particularly of the XRete language! ([FL91, Tho91a, Tho91b, Fag87] used for writing real-time embedded merge applications data interpretation or interpretation (SF88) XRete combines the use of rules of production and rules of deduction whose logical semantics are described by elsewhere [FL91, Fag90b, Fag90a].

The formalization of operational semantics makes it possible to describe mathematically running programs, defining and validating compilation algorithms or study for example their parallelization, The purpose of this article is to give a complete operational semantics of production systems and to derive from them effective Timplémentation via the Rete algorithm used. Such an approach provides a proof of the compiler that performs these transformations in practice. We show also that the Treat algorithm proposed by D. Miranker as an alternative to Rete [Mir90] can be defined using the same method by a simple modification of the formal system presented, This method is sufficiently general to be used both for parallel and sequential implementations. On the other hand, we will see that the optimizations of Rete generally proposed (hash-code, permutation single and multi-schema tests [GD84]) are easily defined in this formalism.

The plan is as follows: in a first part we briefly present the languages & rules of production, and give their operational semantics. We then derive by successive refinements from this operational semantics an algorithm (Rete [For82]) that implements it; we give proof correction of this algorithm, in its sequential and parallel versions. We let's end with a description of the different existing and possible implementations of these languages.

2 Operational semantics of production systems

2.1 Production systems

A production system, denoted $\langle W, RB\rangle$, consists of a **rule-base** RB, also called program, and a base of facts or objects W, also called **working memory**.

The ensemble of facts / objects will be identified in this article with the **Herbrand base** B_H (ie. all closed atomic propositions) formed on an alphabet of symbols of constants, functions, and first-order predicates.

A production rule is a triple:

$$\pi \rightarrow \alpha, \rho$$

where π expresses a condition on the facts-base, α represents a series of facts **added** and ρ a series of facts **removed**. More precisely π, α and ρ can be identified with sequels of literals of first-order logic, that is, atomic propositions (called **schemas**), or the negation of atomic propositions, which may contain variables. A production rule has the following form:

$$A_1, ..., A_k, \neg B_1, .., \neg B_l \rightarrow C_1, ...C_m, \neg D_1, ..., \neg D_n$$

with the generally supplementary restriction that the variables of the schemas B_j, C_j, D_j appear in the A_i's.

"Negation by default" means to take $\neg A$ true if no instantiation of A is in the facts-base. An **instantiation** of a rule is a k-tuple of **working memory elements**, or WMEs, $(o_1, ..., o_k)$ satisfying the left hand side of the rule, that is to say, there exists a substitution σ of the variables of the rule verifying $A_i\sigma = o_i$, for $1 \leq i \leq k$,

and $\neg B_j\sigma$, for $1 \leq j \leq l$. The negation on the right side of the rule represents the action of deleting a fact ($\neg D_i\sigma$) from working memory.

The production rules allow expressing very general transformations fact base while retaining an attractive declarative character. for example the simplest sorting algorithm that can be imagined: iteratively couples of badly ordered elements, written directly with a single rule of production:

```
rule tri
?x : (elem indice=?i val=?v)
?y : (elem indice=?j &> ?i val=?w &< ?v)
->
modify (?x indice=?j)
modify (?y indice=?i)
endrule
```

where variables are prefixed with ? (XRete syntax), the filtering in schemas is done by the name of the attributes, and "modify" is equivalent to an addition of the fact followed by deletion. When no more rules are triggerable, the stable state obtained (the fixed point of the program) corresponds to a sorted configuration. The different exchange sorting algorithms are obtained by specifying the triggering order rules by a priority mechanism for example. It is remarkable that one in this way obtain optimal complexity sorting algorithms on average in $O(n \log n)$ where n is the number of elements to be sorted, applying the techniques described in this article (see section 4.2.1).

The various control strategies of the triggering order of the production rules (the strategies *lex, mea* [For81], static priorities, dynamics, meta-rules [Tho91a], etc ...) are not covered in this article. We consider the production systems as non-deterministic systems.

2.2 Inference Cycle

The content of the working memory represents the state of the system at a given moment of computation, as the value of the variables represents the state of the execution of a program in a classical language. The rules are analogous to the "program" which defines the changes made to the working memory. The formal definition of inference cycle requires formally defining the filtering phase, i.e. searching for rule instantiations.

A condition of the rule π consists of a sequence of literals L. So we have $\pi = \pi'.L$ or $\pi = true$. We define the relation of **satisfaction** of a condition π on W by a **partial instantiation** $(o_1, ..., o_k)$, denoted $(W, o_1, ..., o_k) \vDash \pi$, by the following system:

$$W \vDash true \tag{1}$$

$$\frac{(W, o_1, ..., o_{k-1}) \vDash \pi \quad o_k \in W \quad p(o_1, ..., o_k)}{(W, o_1, ..., o_k) \vDash \pi.P} \tag{2}$$

$$\frac{(W, o_1, ..., o_k) \vDash \pi \quad \neg\exists o \in W \quad p(o_1, ..., o_k, o)}{(W, o_1, ..., o_k) \vDash \pi.P} \tag{3}$$

where $p(o_1, ..., o_j)$ denotes the conjunction of tests on $(o_1, ..., o_j)$ appearing in the **schema** P.

A **instantiation** of rule of $\langle W, RB\rangle$ is a pair $(R, (o_1, ..., o_{ar(R)}))$ with $R \in RB(o_1, ..., o_n) \in W^n$ such that $(W, o_1, ..., o_n) \vDash \pi(R)$. Each instantiation of rule defines a transition on the working memory.

$$W \rightarrow_{(R,(o_1,...,o_n))} (W\ \rho(R, o_1, ..., o_n)) \cup \alpha(R, o_1, ..., o_n)$$

The non-deterministic transition associated with the rule base RB, denoted by $\rightarrow$, constitutes the **cycle of inference**. It is defined by:

$$\frac{R \in RB(W, o_1, ..., o_n) \vDash \pi(R) \quad W \rightarrow_{(R,(o_1,...,o_n))} W'}{RB \vdash W \rightarrow W'} \tag{4}$$

Figure 1 schematically shows the inference cycle. [See original paper for figures]

Figure 1:

Filtering

$$\pi(R)(W, o_1, ..., o_n) = true$$

Selection SEL

$$(R, (o_1, ..., o_n))$$

Applying

$$W \rightarrow_{(R,(o_1,...,o_n))} W'$$

2.3 Operational Semantics

The operational semantics of a program $\langle W, RB \rangle$ is an **ensemble of stable states** defined as the ensemble of terminal states for the relation $\rightarrow$, that is to say:

$$||\langle W, RB \rangle|| = \{W_s \in \mathcal{P}(B_H) | RB \vdash W \rightarrow^* W_s \\ \text{and} \quad RB \vdash W_s \rightarrow W' \Rightarrow W_s = W'\}$$

where the relation $\rightarrow^*$ is the transitive reflexive closure of the relation $\rightarrow$:

$$\frac{RB \vdash W \rightarrow^* W}{RB \vdash W \rightarrow^* W''} \quad \frac{RB \vdash W \rightarrow^* W' \quad RB \vdash W' \rightarrow W''}{RB \vdash W \rightarrow^* W''}$$

3 Compilation

3.1 Rete algorithm: construction of the Rete graph

The Rete [For82] algorithm is an efficient algorithm that calculates all instantiations of rules incrementally. It was developed for OPS5. It is based on two principles:

1. Memorization: it is assumed that the number of modifications made to the working memory at each inference cycle is small compared to the size of the working memory. This assumption is verified in practice in many rule bases [GF83]. Therefore, all instantiations of rules in cycle n is relatively little different from its state in cycle $n-1$. So Rete's idea is to calculate only the modifications of the set of rule instantiations from one cycle to another. For this, the intermediate calculations (the partial rule instantiations) are stored.

2. Sharing: two rules can have some of their common conditions. The calculation relating to this sub-condition is factored between the two rules.

Rete has been widely used for the efficient implementation of rule-based languages.

We derive from equations (2) and (3) a formal presentation of this algorithm.

First, the **predicate** $p(o_1, ..., o_j, o)$ associated with schema P in equations (2) and (3) can be broken down into two parts:

$$p(o_1, ..., o_j, o) \equiv q(o) \wedge r(o_1, ..., o_j, o)$$

where $q(o)$ denotes the conjunction of the **mono-schema tests** on o appearing in the schema P, and $r(o_1, ..., o_j, o)$ denotes the conjunction of **multi-schema tests** between $(o_1, ..., o_j)$ and o appearing in the schema P_i. The decomposition of the test p into q and r does not make any logical sense, but it permits to reduce the combination of the search for n-tuples $(o_1, ..., o_j, o)$ that satisfy p because we restrict the search for o on objects verifying q.

This allows us to transform equations (2) and (3) into:

$$\frac{(W, o_1, ..., o_{k-1}) \vDash \pi \quad q(o_k) \quad r(o_1, ..., o_k)}{(W, o_1, ..., o_k) \vDash \pi.P} \tag{5}$$

$$\frac{(W, o_1, ..., o_k) \vDash \pi \quad \neg\exists o \in W \quad q(o) \wedge r(o_1, ..., o_k, o)}{(W, o_1, ..., o_k) \vDash \pi.\neg P} \tag{6}$$

The incremental character of the Rete algorithm comes from the storage of the n-tuples (respectively objects) that verifies the tests π (respectively q) in the **left memory** lm (respectively the **right memory** rm).

So, for a condition:

$$\pi = P_1, ..., P_n$$

the following left and right memories lm and rm are defined*:

Definition 1

$$lm_i(E) = \{(o_1, ..., o_{k_i}) \in E^{k_i} | ((o_1, ..., o_k), E) \vDash \pi_i\} = E/\pi_i$$

$$rm_i(E) = \{o \in E | q_{i+1}(o)\} = E/q_{i+1}$$

Equations (1), (5) and (6) naturally become:

$$\frac{(o_1, ..., o_{k-1}) \in lm_{i-1}(W) \quad o_k \in rm_{i-1}(W) \quad r_{i-1}(o_1, ..., o_k)}{(o_1, ..., o_k) \in lm_i(W)} \tag{7}$$

for a positive schema and $i \neq 1$, and:

$$\frac{(o_1, ..., o_k) \in lm_{i-1}(W) \quad \neg\exists o \in rm_{i-1}(W) \quad r_{i-1}(o_1, ..., o_k, o)}{(o_1, ..., o_k) \in lm_i(W)} \tag{8}$$

for a negative schema and $i \neq 1$, and:

$$lm_1(W) = rm_0(W) \tag{9}$$

*lm is called β-memories in the terminology of OPS5. rm is called α-memories (?) [YKY: second footnote is unreadable]

We now define the ensemble $rulem$ that contains the triggerable rule instantiations.

$$rulem(R, W) = \{(R, (o_1, ..., o_n))|(o_1, ..., o_n) \in lm_n(W)\} \tag{10}$$

We can now write equation (4) as:

$$\frac{(o_1, ..., o_k) \in rulem(R) \quad W \rightarrow_{(R,(o_1,...,o_k))} W' \quad R \in RB}{W \rightarrow W'} \tag{11}$$

The Rete algorithm uses a graph structure.

For each left and right **pile** lm and rm is associated a node of the graph identified by the pile to which it is attached.

On the other hand, we associate with each of the equations (7) and (8) a node called **juncture**. The junctures corresponding to equation (7) are said to be positive; those corresponding to equation (8) are said to be negative. At each of these nodes are associated three **piles**: the **left** pile $lm_{i-1}(W)$, the **right** pile $rm_{i-1}(W)$ and the **resultant** pile $lm_i(W)$.

By connecting with an arc the nodes joining the nodes that the piles are associated with, we obtain a graph. By orienting the arcs of the junctures towards the resultant piles, and the left and right piles towards the junctures, we obtain an oriented graph (figure 2).

Figure 2. (Oriented rete graph)

Calculation of $\pi(o_1, ..., o_n)$:

3.2 Rete algorithm: execution of the graph

Formally, Rete associates an agent with each node of its graph. Schematically, this agent corresponding to a juncture is responsible for maintaining the contents of the resultant memory lm_i depending on the evolution of the left and right memories lm_{i-1} and rm_{i-1}.

These agents communicate by sending messages, called tokens, of the form $\langle [+-], e \rangle$. The element e is either an object or an n-tuple. The sign + or - means we want to add or remove the element. It corresponds to the two basic actions defined in the right hand sides of rules.

The agents communicate with each other by synchronous communications. Arcs of the graph are the communication channels of these messages. The communications are in the direction of the orientation of the arcs.

We assume that there is a special Root agent responsible for receiving and distributing messages received from the outside I In retracts an object o from the working memory sends [... unreadable ...]

We give the operational semantics of these agents.

For each agent, the rule is of the form

$$m? \xrightarrow{a} ms! \tag{12}$$

where $m?$ is the message (token) received, $ms!$ the list of tokens sent in response and a the eventual action executed.