Theoretical Computer Science 71 (1990) 177–191
North-Holland

PRINCIPLES OF PLASMA PATTERN AND ALTERNATIVE STRUCTURE COMPILATION

Jean-Paul ARCANGELI and Christian POMIAN

Laboratoire Langages et Systèmes Informatiques, Université Paul Sabatier, 118 Route de Narbonne, 31062 Toulouse Cedex, France

Abstract. Compilation is another strategy for pattern matching implementation. In a first step, patterns are translated in a program in which backtracking is performed by simple jumps. Then, at run time, only message parsing is done. Code for a pattern is inductively built from elementary pattern codes according to the syntax. A formal model describes our compilation process and leads us to implement a single pass compiler.

1. Introduction

Plasma is a lambda-language based on the message-passing concept. It is a sequential implementation of the actor model of computation defined by C. Hewitt. In this model, actors are independent entities communicating by message-passing and a set of actors is presented as a society of cooperating experts for problem-solving [10].

Plasma differs from other applicative languages like Lisp by some original features like pattern-matching, closures and continuations. Pattern-matching and closures create environments. A receiver actor is a closure. It is both composed by a "definition environment" representing its acquaintances about the external world, and by a "script" which describes the actor's behavior. The script is also composed by a "pattern" which is a symbolic representation of the set of messages that the actor can accept and by a "body" describing the actor's answer to the accepted message. To decide if an actor may accept or reject the message that it receives, a well-known technique is used, called "pattern matching". Plasma pattern matching is a very powerful mechanism allowing both to perform message selection and to build an environment in which the message matches the pattern. If the message sent matches the pattern, pattern matching succeeds and the actor's body is evaluated in the definition environment increased by the resulting pattern matching environment, otherwise it fails. Plasma pattern matching is presented in Section 2.

Plasma pattern matching subsumes parameter binding but also the alternative structure. The actor "cases" is composed by a succession of receivers actors which patterns are allowed to overlap. The condition is a message sent to the actor cases and filtered by each receiver from top to bottom until it matches one's pattern. Then, the corresponding body is evaluated.

Several implementations of Plasma have been realized [6, 18] and a virtual machine, called LILA (Language for Implementation of Applicatives Languages), was designed to support the Plasma interpreter and to subsume portability [11, 19]. The framework of this work is the definition and the implementation of an efficient, complete and portable system for actor programming. Thus, a concurrent implementation, called AL1, of the actor model has been realized [14] and a compiler for Plasma actors is currently being developed and is devoted to be extended to the concurrent system. So, the pattern compiler is one important component of this Plasma compiler. We present in this paper the basic principles for achieving Plasma pattern compilation.

For us, pattern compilation means a translation of an expression in a language into a semantically equivalent expression in a target intermediate language. Our translation process transforms any Plasma pattern into a LILA program. To keep this presentation quite clear, we shall not give the resulting code in LILA but in a sequential pseudo-code like an assembly language, with conditionals and jumps to symbolic labels, manipulating a dotted-pair memory (that can be straight rewrited in LILA or even in Lisp). Moreover, we shall only describe the translation process for main Plasma patterns and we shall not deal with particular Plasma patterns such as anonymous, logical or applying predicates patterns. Of course, our implementation deals with these patterns.

In Section 3, we discuss pattern matching implementation. We present the principal ideas of our method to compile a sequence (Section 4), an intersection (Section 5) and a union (Section 6) of patterns; we show how compiled code results from a simple composition of each sub-pattern code, and that compilation of patterns also assumes the alternative structure compilation. We show, in Section 7, how a single pass compiler results from a formal model describing our method. In conclusion, we discuss code efficiency and optimizations.

This paper is a synthesis of our works about compilation of Plasma patterns [2–4]. About compilation of patterns, we must mention works of Emanuelson and Haraldsson (Lisp code for a pattern is obtained by partial evaluation of a Lisp pattern matching interpreter [7, 8]), a method for compilation of pattern matching in case-expression [5] and for Miranda alternative structure [17].

2. Plasma pattern matching

Plasma pattern matching is a second order semi-unification. It is an asymmetrical unification process (semi-unification) taking two terms: a pattern with variables and a message which does not contain any variable.

Plasma patterns are inductively built from elementary patterns using the syntactic operators brackets, AND and OR. Brackets define an ordered sequence of elements: it is the basic Plasma structure analogous to the Lisp list structure. Elementary patterns are constants, simple or segment variables. If $P1, \ldots, Pn$ are patterns,

[*P*1...*Pn*] is the ordered sequence, (AND *P*1...*Pn*) the intersection and (OR *P*1...*Pn*) the union of *P*1, ..., *Pn*.

A message can be (inductive definition) a constant or a sequence of messages. A pattern can be an elementary or a composed pattern. Let us give a grammar for main patterns:

(1) ⟨Elementary-Pattern⟩ → constant
(2) ⟨Elementary-Pattern⟩ → :identifier
(3) ⟨Elementary-Pattern⟩ → !:identifier
(4) ⟨Composed-Pattern⟩ → [⟨Sequence-Pattern⟩]
(5) ⟨Composed-Pattern⟩ → (AND ⟨And-Pattern⟩)
(6) ⟨Composed-Pattern⟩ → (OR ⟨Or-Pattern⟩)
(7) ⟨Sequence-Pattern⟩ → ⟨Composed-Pattern⟩⟨Sequence-Pattern⟩
(8) ⟨Sequence-Pattern⟩ → ⟨Elementary-Pattern⟩⟨Sequence-Pattern⟩
(9) ⟨Sequence-Pattern⟩ → 1
(10) ⟨And-Pattern⟩ → ⟨Composed-Pattern⟩⟨And-Pattern⟩
(11) ⟨And-Pattern⟩ → 1
(12) ⟨Or-Pattern⟩ → ⟨Composed-Pattern⟩⟨Or-Pattern⟩
(13) ⟨Or-Pattern⟩ → 1

Remark. Syntactically, ⟨And-Pattern⟩ and ⟨Or-Pattern⟩ are equivalent. But, we give for them different productions because we further attach different semantic rules to these productions.

A simple variable is denoted by :identifier and a segment variable by !:identifier.

Any variable can be repeated in the pattern. Segment variables can be linked with an any-length segment of the message (second order). So, pattern matching is very powerful but also complex and expensive because segment variables involve non-determinism in the search of the environment.

Definition. A *substitution S* is a set of bindings between variables of a pattern and non-overlapping parts of a message. The corresponding *substitution function S* is such as *S*(Pattern) = Message. Pattern matching function can be considered as the reciprocal function of the substitution [6].

Definition. Pattern matching succeeds if and only if such a substitution exists otherwise it fails.

Definition. The *distinguished short left substitution* is the first substitution obtained by traversing both the pattern and the message from left to right [9].

Definition. The *resulting environment* of the Plasma pattern matching is the distinguished short left substitution.

We are now going to present what is matching a composed pattern. In the next definitions, *Pi* is a pattern, ∀*i*.

Definition. A message M *matches* a sequence of patterns $[P1...Pn]$ iff

$$\exists S / [S(P1)S(P2)...S(Pn)] = M.$$

Many pattern matching examples are presented in [4, 10, 12]. We only want to show here the great power of expression that the use of segment variables offer and how they allow any complex data structure to be easily decomposed.

Example. [1 2 3] matches $[:A\ !:B]$ and the resulting environment is $\{(A.1)(B.[2\ 3])\}$. $[:A\ !:B]$ decomposes a sequence in *car* and *cdr*.

Example. [1 2 3] matches $[!:A\ :B]$ and gives $\{(A.[1\ 2])(B.3)\}$. B is bound with the last element of the message.

Example. [4 5 6 5 4] matches $[!:A\ :D\ !:B\ :D\ !:C]$. Pattern matching resulting environment is the short left substitution $\{(A.[\ ])(D.4)(B.[5\ 6\ 5])(C.[\ ])\}$. Another substitution is $\{(A.[4])(D.5)(B.[6])(C.[4])\}$.

The search of the pattern matching environment is performed by traversing both the pattern and the message from left to right. A segment variable is first bound with [ ] (0-length segment). The process keeps a way to increase the bound segment and resume itself (backtrack point), and goes on with the message and the following of the pattern. If the process further fails, it is resumed at this backtrack point. Otherwise, it stops when a first substitution is found. An algorithmic definition of pattern matching is given in [9, 16, 18].

Example. A last example to show that backtracking must sometimes be performed in sub-patterns: [[1 2 3 4] 2 3] matches $[[!:A\ !:X\ !:B]\ !:X]$ and the matching environment $\{(A.[1])(X.[2\ 3])(B.[4])\}$ is obtained after a long and expensive search.

Remark. The Plasma pattern matching algorithm can easily be extended to compute the distinguished long (last) left substitution or even the set of all substitutions.

Definition. M matches an intersection of patterns (AND $P1...Pn$) iff $\exists S$ a substitution/$\forall i,\ 1 \leqslant i \leqslant n,\ S(Pi) = M$.

Example. $[a]$, $[a\ a]$, $[a...a]$ matches (AND $[:A\ !:B]\ [!:C\ :A]$), $\forall a$.

Definition. M matches a union of patterns (OR $P1...Pn$) iff $\exists i,\ 1 \leqslant i \leqslant n / \forall j,\ 1 \leqslant j < i$, M does not match Pj but matches Pi.

Example. Any sequence of 1, 2 or 3 elements matches (OR $[:A]\ [:A\ :B]\ [:A\ :B\ :C]$). Matching [1 [2 3]] with this pattern produces $\{(A.1)(B.[2\ 3])\}$.

Remark. And and Or operators are not commutative. Matching a message with an intersection of patterns produces the same "set of all substitutions" whatever the order of these patterns in the intersection is. But, the short left substitution, defining

the environment, is not always the same (see the next example). OR patterns are allowed to overlap and so top-down ordering is necessary.

Example

[1 2] matches (AND [!:*A* !:*B*] [!:*B* !:*A*]) and gives {(*A*.[ ])(*B*.[1 2])}
[1 2] matches (AND [!:*B* !:*A*] [!:*A* !:*B*]) and gives {(*B*.[ ])(*A*.[1 2])}
[1 2] matches (OR [:*A* :*B*] [:*A* !:*B*]) and gives {(*A*.1)(*B*.2)}
[1 2] matches (OR [:*A* !:*B*] [:*A* :*B*]) and gives {(*A*.1)(*B*.[2])}.

3. Implementation of the pattern matching process

As the grammar for patterns defines a programming language, a pattern can be regarded as a program and a message as a data to be applied to this program. Thus, two modes of implementation are possible.

In the classical implementation, the pattern matching process is an interpreter, parsing both the pattern and the message to compute the environment. A stack is useful to save any context that could eventually be necessary to resume the process if it fails. For each point of choice, the context which must describe the pattern matching current state (remainder of the pattern, remainder of the message and current environment), is pushed on the stack. Thus, backtracking is performed by restoring the last context pushed on the stack. Actually, the last context pushed on the stack is the current process failure continuation. The initial context pushed on the stack is the pattern matching failure continuation. The search of the environment corresponds to a depth-first traversal of the search tree.

We propose another strategy: the pattern is given to a pattern compiler which produces a semantically equivalent code. At run time, this code is run and performs the message analysis only. This is a sort of currying process:

Pattern-matching(P, M) = [Pattern-compiler(P)][M].

4. Compiling a sequence of patterns

Code for a pattern must decompose the message in left-right order, in the same way as the pattern matching interpreter. The interpreter preserves all the information about matching by pushing the description of every point of choice on the stack. The stack is cleared out only when a definitive success occurs. When it is fully developed, the stack contains a description of each point of choice. These descriptions are ordered historically from the bottom to the top. The main idea is that the code execution simulates the movement of the stack: information for backtracking is held by the code itself and the structure of the code can be regarded as the image of the fully developed stack [3].

Therefore, code corresponding to a segment variable contains a label where backtracking can be performed. A jump to this label is done when matching fails.

For traversing the message (i.e. implemented like a Lisp *S*-expression) we shall use car and cdr Lisp-like primitives. The constructed environment is represented by a set of variables pointing on the data message. One variable, *A*, corresponds to a pattern simple variable :*A* and two ones, *A*-begin and *A*-end, correspond to a segment variable !:*A*. Let [*M*1...*Mi*...*Mj*...*Mn*] be the message; !:*A* is bound with [*Mi*...*Mj*] iff *A*-begin points on [*Mi*...*Mj*...*Mn*] and *A*-end points on [*Mj*+1...*Mn*]. Therefore, !:*A* is bound with [ ] iff *A*-begin and *A*-end are both pointing on the same element. Pointers on the message are also useful for nested message processing (*M*-cdr) and multiple message analysis (*M*-save).

Thus, compilation changes any Plasma pattern into a set of pointers and into a program in an intermediate language computing their values.

In the following examples, *M* represents the current status of the message, "back-to-?" is the name of the current backtrack label, "failure" is a label where pattern matching failure continuation is performed ("failure" is the initial backtrack label). Matching success continuation is the implicit continuation of the program.

4.1. Code for [...*A*...] (*A is a constant*)

```
...
if M = [ ] goto back-to-?
if car(M) ≠ A goto back-to-?
M := cdr(M)
...
```

4.2. Code for [...:*A*...] (*first occurrence of* :*A*)

```
...
if M = [ ] goto back-to-?
A := car(M)
M := cdr(M)
...
```

4.3. Code for [...:*A*...] (*second occurrence of* :*A*)

```
...
if M = [ ] goto back-to-?
if A ≠ car(M) goto back-to-?
M := cdr(M)
...
```

4.4. Code for [...!:*A*...] (*first occurrence*)

```
...
   A-begin := M                ; Initially A is bound with [ ]
   A-end := M
   goto going-on-A
back-to-A                      ; Matching failed
```

```
    if A-end = [ ] goto back-to-?   ; A-binding is modified if possible
    A-end := cdr(A-end)             ; if not, another backtrack is done
    M := A-end
going-on-A                          ; backtracking must now be
...                                 ; performed at back-to-A
```

4.5. Code for [...CP...] (*where* CP *is a composed pattern*)

```
...
if M = [ ] goto back-to-?
M-cdr := cdr(M)
M := car(M)
...                                 ; code for the nested pattern CP
...                                 ; where M is the message and
...                                 ; where M-cdr does not appear
M := M-cdr                          ; Matching goes on with CP resulting
...                                 ; environment. Here, the backtrack
                                    ; point is the last in CP code.
```

In this way, code for any pattern is linear, even if it is a several level nested pattern.

4.6. Code for any sequence of patterns

A backtrack point in the code is represented by a "back-to-?" label. When pattern matching fails, it must be resumed at the previous point of choice. As code for a pattern must parse the message from left to right, sub-pattern codes must be composed in left-right order. Then, the backtrack point is obviously the previous one in the code. Code is run sequentially and so backtracking is performed by simple backwards jumps to the previous backtrack point in the code.

Corresponding to the syntactic operator [...], the semantic code constructor operator is the left-right concatenation. Code for a sequence of patterns results from the simple concatenation of each pattern code. This concatenation creates a string of backtrack points.

Several examples of generated code are given in [3] and [4]. We only give one simple example here to show code composition.

Example. Code for [!:*A* :*B*]

```
begin
  if M is not a sequence goto failure
  A-begin := M                  ; □
  A-end := M                    ; □
  goto going-on-A               ; □
back-to-A                       ; □ Code
```

```
  if A-end = [ ] goto failure  ; □ for !:A
  A-end := cdr(A-end)          ; □
  M := A-end                   ; □
going-on-A                     ; □
  if M = [ ] goto back-to-A    ; ◇
  B := car(M)                  ; ◇ Code for :B
  M := cdr(M)                  ; ◇
  if M ≠ [ ] goto back-to-A
end.
```

5. Compiling an intersection of patterns

To match (AND $P1...Pn$), as a variable can appear both in $Pi+1$ and in (AND $P1...Pi$), $\forall i$, matching $Pi+1$ begins with (AND $P1...Pi$) pattern matching resulting environment. If it fails, the process must be resumed at the last backtrack point in (AND $P1...Pi$).

As AND is not commutative, code for an intersection of patterns must be the left-right concatenation of each pattern code. Then, in the same way as for a sequence of patterns, initial backtracking in $Pi+1$ code is performed by a jump to the last backtrack point in Pi code.

AND semantic operator performs left-right concatenation but also adds statements to save and restore the message.

Example. Code for (AND $P1$ $P2$). Let $P1$ and $P2$ be two patterns and "back-to-1" the last backtrack point in $P1$ code.

```
begin
  M-save := M          ; M is preserved
  ...                  ; □
back-to-1              ; □ Code for P1
  ...                  ; □
  M := M-save          ; M is restored
  ...                  ; ◇
  if...goto back-to-1  ; ◇
  ...                  ; ◇ Code for P2
  if...goto back-to-1  ; ◇
  ...                  ; ◇
end.
```

6. Compiling a union of patterns

6.1. *Code for* (OR $P1...Pn$)

$\forall i$, if matching Pi fails, the interpreter tries to match $Pi+1$. As code for a union of patterns results, still for the same reasons, from the left-right concatenation of

each pattern code, initial backtracking in *Pi* is performed by a forward jump to the beginning of *Pi*+1 code. So, *Pi* failure continuation is *Pi*+1 code beginning, and *Pi* success continuation is "success", a label where the matching success continuation is performed.

Thus, OR semantic operator performs left-right concatenation, message preserving, but also implements the success continuation by adding a label and appropriate jumps.

Compiling [(OR *P*1 *P*2) *P*3] sets problems [4] but can be rewritten in (OR [*P*1 *P*3] [*P*2 *P*3]) and then compiled.

Example. Code for (OR *P*1 *P*2).

```
begin
   M-save := M        ; M is preserved
   ...                ; □
   if...goto begin-2  ; □
   ...                ; □ Code for P1
   if...goto begin-2  ; □
   ...                ; □
   goto success
begin-2
   M := M-save        ; M is restored
   ...                ; ◇
   if...goto failure  ; ◇
   ...                ; ◇ Code for P2
   if...goto failure  ; ◇
   ...                ; ◇
success
end.
```

6.2. Code for (CASES (*P*1 Body1)...(*Pn* Body*n*))

The alternative structure can be compiled like a union of patterns. It differs only from the matching success continuation of *Pi* which is the beginning of Body*i* code. A new semantic operator for cases code construction assumes this modification.

7. Implementation of the compiler

As patterns are inductively built from elementary patterns using the syntactic operators brackets, AND, OR and cases, code for a pattern is inductively built from elementary pattern codes according to the syntactic structure. A semantic operator is associated with each syntactic operator, and performs a composition of code based on left-right concatenation.

Because of backtrack points and repetition of variables, sub-patterns must be compiled in left-right order. Resulting sub-codes are then composed as previously described. However, we think that a preparatory pass on the pattern could make sub-pattern compilation processes independent. Then, we can imagine that sub-patterns could be compiled in parallel. This can be an important property if several passes are useful (see Sections 8.2 and 8.3).

7.1. First implementation

In our first implementation, the pattern compiler is derived from a pattern matching interpreter. There, message parsing actions are replaced by generating code actions. Therefore, we define a pattern compiler that performs a single top-down analysis of the pattern, as the interpreter [15]. This compiler is a Plasma actor, using pattern matching to decompose the pattern [2, 3].

7.2. A formal model

The translation mechanism can be described using a formalism well-adapted to compilation, the attribute formalism [1, 13]. Compilation consists of annotating each node of the pattern syntactic tree with semantic quantities by computing semantic rules.

Our model is given in the Appendix. As inherited attribute values depend only on the values of the left siblings at a node, our grammar is left attributed. Left attributed grammars have the important property that one depth-first traversal of the parse tree is enough to compute the attribute values [1]; semantic rules can be evaluated during parsing and so our model defines a single-pass compiler [4].

7.3. Second implementation

FNC is a metacompiler operating on the class of strongly non-circular attributed grammars. It takes a grammar for patterns, a set of attributes and a set of semantic rules, and produces a Lisp program which performs the single-pass compilation of patterns.

7.4. Extensions

As code for patterns is obtained in a very modular way, the pattern compiler maintenance is easy; it can be extended to deal with particular Plasma patterns like anonymous variables or applying predicate patterns. Moreover, it could be easily modified to produce a code computing the long left substitution or all the substitutions.

8. Code efficiency and optimizations

8.1. Code efficiency

Code is inexpensive in storage space: it needs only one pointer for a simple variable and two ones for a segment variable (it also needs a pointer for each level

of nested pattern). Pointers are stored into registers and so running code does not involve the target machine garbage collector.

To perform pattern matching, only message parsing is done. No other recursive call is necessary and the stack useful for the interpreter can be removed. The process can easily be resumed by doing simple jumps in a linear code (compilation transforms any nested structure into a linear one).

Realized tests have shown that compiled pattern matching runs generally 4–8 times faster than interpreted pattern matching. Nevertheless, backtracking in a sequence of patterns and cases pattern matching can sometimes be improved.

8.2. Backtracking optimization

First, we want to avoid generation of unnecessary backtrack points; when a segment variable is the last element of a sequence, no backtrack point is generated.

Example. Code for [:*A* !:*B*]

```
begin
  if M is not a sequence goto failure
  if M = [ ] goto failure                ; □
  A := car(M)                            ; □ Code for :A
  M := cdr(M)                            ; □
  B-begin := M                           ; ◇ Code
  B-end := nil                           ; ◇ for !:B
end.
```

Then, we remark that a jump to the previous backtrack label in the code of a sequence of patterns is not always the most efficient way to perform backtracking.

Example. Let [!:*X* [!:*Y* !:*Z*] 3 !:*T*] be a pattern. Efficient backtracking for 3 or !:*T* must be performed by a jump to back-to-*X* and not by a jump to back-to-*Y*.

In a sequence of patterns, efficient backtracking for a constant, a sequence of patterns or even a first occurrence of a variable, must be performed by a jump to the backtrack point corresponding to the left-most variable among the previous segment variables at the same level. Backtracking for a second occurrence of a variable must be performed by a jump in the linear code to the same backtrack label as the first occurrence, however the first occurrence is nested. Implementing this improvement would change our method for backtracking compilation a lot; for any pattern, the corresponding backtracking label would be computed in a preliminary pass before code generation.

8.3. Cases optimization

Conditional processing consists of matching patterns in turn until one succeeds. This can be improved by collecting information during the failing processes to avoid

repetition of the same tests. An alternative structure is composed by an ordered succession of patterns. There, patterns beginning in the same way are grouped together [17]. Each group is an ordered succession of overlapping patterns. Whereas a union of patterns is not commutative, two adjacent groups can be exchanged because they do not overlap. This can allow exchanged groups to be grouped together with their new adjoining group. Then, the common beginning is factorized and the process is repeated in all the groups. When the alternative structure is wholly rewritten in such a manner, it is compiled as previously described.

Example.

(OR 1 [:*X* !: *Y* :*X* !:*Z*] [ ] [:*X* !: *Y* 1 2])
⇒ (OR 1
(OR [:*X* !: *Y* :*X* !:*Z*] [ ] [:*X* !: *Y* 1 2]))
⇒ (OR 1
[(OR ![:*X* !: *Y* :*X* !:*Z*] ![ ] ![:*X* !: *Y* 1 2])])

![...] denotes a succession (but not a sequence) of patterns

⇒ (OR 1
[(OR ![ ]
(OR ![:*X* !: *Y* :*X* !:*Z*] ![:*X* !: *Y* 1 2]))])
⇒ (OR 1
[(OR ![ ]
![:*X* (OR ![!: *Y* :*X* !:*Z*] ![!: *Y* 1 2])])])

The process is stopped, even if [!: *Y* :*X* !:*Z*] and [!: *Y* 1 2] begins in the same way. Segment variables cannot be factorized: rewriting (OR [:*X* !: *Y* :*X* !:*Z*] [:*X* !: *Y* 1 2]) in [:*X* !: *Y* (OR ![:*X* !:*Z*] ![1 2])] and compiling it, would change the semantic because backtracking in ![:*X* !:*Z*] would be falsely performed in ![1 2] (if a union of ![...] is compiled like any union of sequences). Matching this pattern with [2 1 2] would give {(*X*.2)(*Y*.[ ])} instead of {(*X*.2)(*Y*.[1])(*Z*.[ ])}. Thus, regrouping concerns only univocal patterns.

Appendix

List of attributes:
- SCODE synthesized, code at a node
- HLAB inherited, intermediate attribute for label manipulation
- HSYMB inherited, intermediate attribute for symbol manipulation
- SBACK synthesized and HBACK inherited, current backtrack label
- SSET synthesized and HSET inherited, set of variables currently compiled.

List of functions:
- G1 generates code for a constant as described in Section 4.1
- G2 generates code for a simple variable as described in Sections 4.2 and 4.3
- G3 generates code for a segment variable as decribed in Section 4.4
- || performs code concatenation
- Newlabel creates a new label
- Newsym creates a new symbol
- Union performs union upon sets.

Abbreviations:
- EP for Elementary-Pattern
- CP for Composed-Pattern
- SP for Sequence-Pattern
- AP for And-Pattern
- OP for Or-Pattern.

Semantic Rules:

(1) EP.SCODE := G1(constant, EP.HBACK)
EP.SBACK := EP.HBACK
EP.SSET := EP.HSET

(2) EP.SCODE := G2(identifier, EP.HSET, EP.HBACK)
EP.SBACK := EP.HBACK
EP.SSET := Union({identifier}, EP.HSET)

(3) EP.SCODE := G3(identifier, EP.HSET, EP.HBACK, EP. SBACK)
EP.SBACK := Newlabel
EP.SSET := Union({identifier}, EP.HSET)

(4) CP.SCODE := {if M is not a sequence goto CP.HBACK} || SP.SCODE ||
{if $M \neq [\,]$ goto SP.SBACK}
CP.SBACK := SP.SBACK
SP.HBACK := CP.HBACK
CP.SSET := SP.SSET
SP.HSET := CP.HSET

(5) CP.SCODE := {AP.HSYMB := M} || AP.SCODE
AP.HSYMB := Newsym
CP.SBACK := AP.SBACK
AP.HBACK := CP.HBACK
CP.SSET := AP.SSET
AP.HSET := CP.HSET

(6) CP.SCODE := {OP.HSYMB := M} || OP.SCODE || {OP.HLAB}
OP.HLAB := Newlabel
OP.HSYMB := Newsym
OP.HBACK := CP.HBACK
OP.HSET := CP.HSET

(7) SP1.SCODE := {if M = [ ] goto SP1.HBACK; SP1.HSYMB := cdr(M);
M := car(M)} || CP.SCODE ||
{M := SP1.HSYMB} || SP2.SCODE
SP1.HSYMB := Newsym
SP1.SBACK := SP2.SBACK
CP.HBACK := SP1.HBACK
SP2.HBACK := CP.SBACK
SP1.SSET := SP2.SSET
CP.HSET := SP1.HSET
SP2.HSET := CP.SSET

(8) SP1.SCODE := EP.SCODE || SP2.SCODE
SP1.SBACK := SP2.SBACK
EP.HBACK := SP1.HBACK
SP2.HBACK := EP.SBACK
SP1.SSET := SP2.SSET
EP.HSET := SP1.HSET
SP2.HSET := EP.SSET

(9) SP.SCODE := { }
SP.SBACK := SP.HBACK
SP.SSET := SP.HSET

(10) AP1.SCODE := CP.SCODE || {M := AP1.HSYMB} || AP2.SCODE
AP2.HSYMB := AP1.HSYMB
AP1.SBACK := AP2.SBACK
CP.HBACK := AP1.HBACK
AP2.HBACK := CP.SBACK

(11) AP.SCODE := { }
AP.SBACK := AP.HBACK
AP.SSET := AP.HSET

(12) OP1.SCODE := CP.SCODE || {goto OP1.HLAB; CP.HBACK;
M := OP1.HSYMB} || OP2.SCODE
CP.HBACK := Newlabel
CP.HSET := OP1.HSET
OP2.HLAB := OP1.HLAB
OP2.HSYMB := OP1.HSYMB
OP2.HBACK := OP1.HBACK
OP2.HSET := OP1.HSET

(13) OP.SCODE := {goto OP.HBACK}.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers, Principles, Techniques and Tools* (Addison-Wesley, Reading, MA, 1986).

[2] J.-P. Arcangeli and C. Pomian, Compilation des filtres Plasma en Plasma, Rapport Interne No. 277, Université P. Sabatier, Toulouse, 1987.

[3] J.-P. Arcangeli and C. Pomian, Compilation de plasma: les filtres, *Bigre Globule* **59** (1988) 140-149.

[4] J.-P. Arcangeli and C. Pomian, Plasma pattern compilation: a formal model and some examples, in: T. O'Shea and V. Sgurev, eds., *Artificial Intelligence 3, Methodology, Systems, Applications* (North-Holland, Amsterdam, 1988) 111-120.

[5] L. Augustsson, Compiling pattern matching, in: J.-P. Jouannaud, ed., *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201** (Springer, Berlin, 1985) 368-381.

[6] J.-L. Durieux, Sémantique des liaisons nom-valeur; application à l'implémentation de lambda-langages, Thése d'état, Université P. Sabatier, Toulouse, 1981.

[7] P Emanuelson and A. Haraldsson, On compiling embedded languages in Lisp, in: *Conference Record of the 1980 LISP Conf.* (1980) 208-215.

[8] P. Emanuelson, From abstract model to efficient compilation of patterns, in: M. Dezani-Ciancaglini and U. Montanari, eds., *Int. Symp. on Programming*, Lecture Notes in Computer Science **137** (Springer, Berlin, 1982) 91-104.

[9] M. Gengler, Un algorithme optimisé de semi-unification du second ordre sub arborescences, *Bigre Globule* **59** (1988) 238-[illegible].

[10] C. Hewitt, Viewing control structures as patterns of passing messages, *J. Artificial Intelligence* **8** (1977) 323-364.

[11] D. Julien, Etude et réalisation de la machine virtuelle LILA adaptée à l'écriture d'interprètes, Thèse de 3ème cycle, Université P. Sabatier, Toulouse, 1985.

[12] G. Lapalme and P. Salle, Présentation du langage Plasma version 88, Rapport Interne, Université P. Sabatier, Toulouse, 1988.

[13] C. Livercy, *Théorie des Programmes. Schémas, Preuves, Sémantique* (Dunod Informatique, Paris, 1978).

[14] A. Marcoux, C. Maurel and P. Salle, AL1: A language for distributed applications, in: *Workshop on the Future Trends of Distributed Computing Systems in the 1990s* (IEEE Computer Society Press, Rockville, MD, 1988) 270-276.

[15] E. Neidl, Etude des relations avec l'interprète dans la compilation de Lisp, Thèse de 3ème cycle, Université Paris VI, 1984.

[16] A. Ouvrard, Implémentation classique du filtrage Plasma Elimination des récursivités implicites du filtrage, Rapport E.N.S.E.E.I.H.T. et D.E.A d'Informatique, Université P. Sabatier, Toulouse, 1984.

[17] P. Wadler, Efficient implementation of pattern-matching, in: S.L. Peyton Jones, ed., *The Implementation of Functional Programming Languages* (Prentice-Hall, New York, 1987) 78-103.

[18] C. Pomian, Contribution à la définition de l'implémentation d'un interprète du langage Plasma, Thèse de 3ème cycle, Université P. Sabatier, Toulouse, 1980.

[19] P. Salle, Le bon usage de la machine virtuelle LILA, Rapport Interne No. 241-A, Université P. Sabatier, Toulouse, 1986.