Partial Evaluation applied to Symbolic Pattern Matching with Intelligent Backtrack*

Partial version presented at Workshop for Static Analysis (Bordeaux France) october 1992.

Christian Queinnec & Jean-Marie Geffroy†
École Polytechnique & INRIA-Rocquencourt

Abstract

Symbolic pattern matching as offered by Lisp dialects allows to scan Sexpressions, to verify their shape and to extract or compare subparts of them. A rich set of patterns exists and among them alternate patterns. Such patterns are handled through backtrack: a failure forces the pattern matcher to regress to its last point of choice and to try another branch. The usual naïve backtrack algorithm forgets all the informations acquired on the datum from the last point of choice up to the failure point. Our algorithm provides intelligent backtrack i.e. these informations are given back to the pattern matcher which can then use them to choose the appropriate backtrack point therefore eliminating redundant tests or dead-end branches.

It turns out that this very simple idea "Don't throw any information painfully acquired", forms the basis of many clever algorithms such as Knuth-Morris-Pratt (KMP) or Boyer-Moore (BM) for strings, Hoffmann-O'Donnell for trees etc. The other benefit comes from the great aptitude of our pattern matcher to be partially evaluated into efficient code: no eurekas are needed to show that we "rediscover" all the previously mentionned algorithms.

The original results of this paper are:

1. a general symbolic pattern matching algorithm with intelligent backtrack. Contrasting with many published pattern matchers, ours offers pattern variables, boolean connectives and unbounded tree-like repetition.
2. we show how failure informations are, most of the time, static (or safely approximated) thus giving great opportunities for our pattern matcher to well behave through partial evaluation.
3. we show that KMP, BM and others are instances of our pattern matcher depending on the exact order (left-to-right, top-to-bottom, etc.) along which composed patterns such as `cons` or `tree` match their sub-patterns.

In conclusion, our pattern matcher appears as the incarnation of a simple but fundamental idea, can be tailored to particular domains such as strings, trees or nucleotides sequences, uses regular partial evaluation to automatically produce superior compilers and eventually provides an interesting framework to teach and compare known algorithms.

Symbolic pattern matching is a popular technique of Lisp or ML families of languages. In these languages, data are usually tree-shaped so that compiling patterns into efficient code is of paramount importance. We based our study on the pattern language presented in [Que90] and its associated denotational semantics. This pattern language allows to express a wide variety of pattern constructions such as boolean composition, segment handling and unbounded repetition. It can express all the usual patterns that may usually be found in Lisp dialects. The pattern language is not user-friendly but is precise and offers only one way to express

*$Revision: 1.17$ of $Date: 1992/11/0607:09:49$, submitted to Journal of Functional Programming.

†Laboratoire d'Informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France. Email: {`queinnec,geffroy`}`@poly.polytechnique.fr`, this work has been partially funded by Greco de Programmation.

patterns; it adheres to the Reduced Instruction Set (RISC) philosophy and therefore deserves to be called an *intermediate language*. The semantics of patterns is expressed through a definitional interpreter which compares a datum and a pattern in a context comprising a success and a failure continuation as well as an environment mapping pattern variables to the data they match. When matching a pattern succeeds, the success continuation is called; conversely a matching failure resolves to call the failure continuation. A direct compiler of this pattern language is also presented in [Que90].

Our first goal was to apply the ideas of partial evaluation, as advocated in [BEJ88, HJ91], to automatically obtain the compiler from the interpreter. Fulfilling this goal offers numerous advantages: *(i)* only the interpreter has to be maintained since the compiler is regenerated from it *(ii)* the interpreter is easier than the compiler to maintain *(iii)* the compiler is ensured to be coherent with the interpreter since partial evaluation is a semantics-preserving transformation.

Our second goal was to extend the definitional interpreter to provide intelligent backtrack. Let us give an example of intelligent backtrack: the composed pattern `(or` ϕ_1 ϕ_2`)` allows to match any expression that can be matched by ϕ_1 or ϕ_2. The naïve interpreter just tries to compare the datum against ϕ_1 with a failure continuation saying that another possibility of success is to match the same datum against ϕ_2. Consider, for example, the following pattern:

`(or (cons` ϕ_3 ϕ_4`) (cons (quote A)` ϕ_5`))`

The pattern `(cons` ϕ_3 ϕ_4`)` matches any binary tree (a `cons` cell or a dotted pair in Lisp terminology) which left branch (or `car`) is matched by ϕ_3 and right branch (`cdr`) is matched by ϕ_4. The constant pattern `(quote A)` can only match an expression equal to `A`.

When compared to a non dotted pair, the first branch of the `or` pattern fails and so will the second branch for the same reason. An intelligent backtrack tries to return to the most recent point of choice which may have an impact on the cause of the failure. In the previous example, a non-`cons` datum will make the intelligent pattern matcher eliminate the last point of choice (the `or` pattern) since the remaining branches cannot be successful.

The essence of our solution is to give back to the failure continuation some informations representing the structure of the datum between the last point of choice and the failure point. *Patterns are clearly the means to describe such informations* so we decide that failure continuations will be invoked on a *description* i.e. a pattern describing what is known so far on the datum. Back to our previous example, the failure against `(cons` ϕ_3 ϕ_4`)` will lead to the description `(not (cons (any) (any)))` meaning that the datum is not a dotted pair. The second branch of the `or` pattern first verifies if the incoming description intersects its pattern; if it is not the case then a failure can be directly generated. This does not require to compare the datum to the pattern `(cons (quote A)` ϕ_5`)` since the datum is matched by the description `(not (cons (any) (any)))` which has an empty intersection with the pattern `(cons (quote A)` ϕ_5`)`. On the other hand, suppose the description to be `(cons (quote A) (any))`, meaning that the datum is a tree which left branch is equal to `A`, then to compare such a datum against the pattern `(cons (quote A)` ϕ_5`)` only requires to check that the right branch of the datum (which is still unknown) is matched by ϕ_5. Statically analyzing the various descriptions that might be produced eliminates redundant tests and propagates failure faster.

Pattern-matching with intelligent backtrack is superior to pattern-matching with blind backtrack. Therefore partial evaluation applied on pattern matching with intelligent backtrack gives superior results. In particular the paper will show that, depending on the exact order along which are handled the `car` and `cdr` part of the `cons` pattern, one can "rediscover" *without any eurekas* Knuth-Morris-Pratt, Aho-Corasick (`car` then `cdr`) or Boyer-Moore, Commentz-Walter (`cdr` then `car`) algorithms.

The paper is organized as follows: we first present a naïve pattern matcher in section 1. Partial evaluation is shortly exposed in section 2. We extend the pattern matcher to handle intelligent backtrack in section 3 then we present an application in the domain of strings in section 4 leading to KMP. We discuss the order of evaluation of the `cons` pattern in section 5 yielding BM behavior. We shortly sketch an application in the domain of trees in section 6. The influence of the initial description is discussed in section 7. A comparison with related works appears in section 8.

$\varepsilon \in$	**Val**	=	The set of data to be matched
$\varphi \in$	**Pat**	=	The set of patterns
$\nu \in$	**Id**	=	The set of identifiers
	Ans	=	The set of final answers
$\rho \in$	**Env**	=	$\mathbf{Id} \rightarrow \mathbf{Val} + \{unbound\}$
$\kappa \in$	**Cont**	=	$\mathbf{Env} \times \mathbf{Alt} \rightarrow \mathbf{Ans}$
$\zeta \in$	**Alt**	=	$\mathbf{Unit} \rightarrow \mathbf{Ans}$
$\mu \in$	**Rep**	=	$\mathbf{Id} \rightarrow \mathbf{Val} \times \mathbf{Env} \times \mathbf{Cont} \times \mathbf{Alt} \rightarrow \mathbf{Ans}$

$$\mathcal{M} : \mathbf{Pat} \rightarrow \mathbf{Val} \times \mathbf{Env} \times \mathbf{Rep} \times \mathbf{Cont} \times \mathbf{Alt} \rightarrow \mathbf{Ans}$$

Table 1: Basic domains

1 A pattern language

Our pattern language is made of a small number of primitives patterns or pattern schemes that can be recursively composed. Both the syntax and the names of the patterns are modeled on Scheme, a dialect of Lisp [IEE91]. Patterns can have components such as labels, immediate values or patterns.

The `(any)` pattern matches anything, `(quote` ε`)` matches only a datum equal to ε, `(cons` ϕ_1 ϕ_2`)` matches any dotted pair which left (`car`) and right (`cdr`) branches are respectively matched by ϕ_1 and ϕ_2. The boolean pattern work as their names suggest it: `(or` ϕ_1 ϕ_2`)` matches anything that can be matched by ϕ_1 or ϕ_2, `(and` ϕ_1 ϕ_2`)` matches anything that is matched by ϕ_1 and ϕ_2 while `(not` ϕ`)` matches anything not matched by ϕ. The most intriguing pattern is `(tree` ν ϕ_1 ϕ_2`)` which expresses the unbounded repetition. `(tree` ν ϕ_1 ϕ_2`)` matches any expression matched by ϕ_2 as well as it matches any expression matched by ϕ_1 where `(hole` ν`)` is equivalent to ϕ_2 or (recursively) to `(tree` ν ϕ_1 ϕ_2`)`. The `(hole` ν`)` pattern can only appear in ϕ_1 and indicates where can appear the repeated pattern ϕ_1 or the terminal pattern ϕ_2. Let us give some examples of the `tree` pattern:

```
(tree a*b (cons (quote A) (hole a*b))
          (cons (quote B) (quote ())) )
```

This pattern matches `(B)`, `(A B)`, `(A A A ...A A B)` i.e. lists containing some `A`s followed by a single `B`.

```
(tree bin (cons (hole bin) (hole bin)) (quote A))
```

This pattern matches `A`, `(A . A)`, `((A . A) . A)` ...i.e. all binary trees with `A` leaves.

```
(tree babar (cons (any) (hole babar))
            (cons B (cons A (cons B (cons A (cons R (any)))))) )
```

This pattern matches any list containing somewhere in it the contiguous symbols `B A B A R`.

Pattern variables are noted `(var` ν`)`. The first occurrence binds the pattern variable ν to the corresponding value. Subsequent occurrences can only match the bound value. For example, a pattern that accepts all non empty lists of equal terms can be specified as:

```
(cons (var term) (tree sequel (cons (var term) (hole sequel))
                              (quote ()) ))
```

The specifications of the naïve pattern-matcher follow conventions of [Sch86][1]. Table 1 shows domains, denotations appear in table 2. Except for `tree` these denotations are simple and quite similar to these already published [Que90, Jør90, Dan91]. The *left* and *right* functions returns the left and right branches of a binary tree, they correspond to the usual `car` and `cdr` of Lisp. Success is marked by an invocation of the success continuation κ on the current failure continuation ζ and on the current environment mapping pattern variables to their value. This allows future computations to "backtrack" i.e. to invoke the failure continuation.

For each pattern ϕ, the $\mathcal{M}$ valuation function associates a λ-term which, given a datum ε, tells if ε can be matched by ϕ. To obtain this boolean answer, the initial call $\mathcal{M}(\phi)(\varepsilon, \rho_{init}, \mu_{init}, \kappa_{init}, \zeta_{init})$ is defined with $\rho_{init} = \lambda\nu.unbound$, $\kappa_{init} = \lambda\rho\zeta.true$ and $\zeta_{init} = \lambda().false$.

[1] The construct **if** c **then** p **else** q is written $c \rightarrow p \,\|\, q$, the rest is in functional style.

$\mathcal{M}[\![\texttt{(any)}]\!] = \lambda\varepsilon\rho\mu\kappa\zeta.\kappa(\rho,\zeta)$

$\mathcal{M}[\![\texttt{(quote}\ \varepsilon\texttt{)}]\!] = \lambda\varepsilon'\rho\mu\kappa\zeta.\ \varepsilon' = \varepsilon \rightarrow\ \kappa(\rho,\zeta)[\!]\ \zeta()$

$\mathcal{M}[\![\texttt{(var}\ \nu\texttt{)}]\!] = \lambda\varepsilon\rho\mu\kappa\zeta.\ \rho(\nu) = \mathit{unbound} \rightarrow\ \kappa(\rho[\nu \rightarrow \varepsilon],\zeta)[\!]\ \ \rho(\nu) = \varepsilon \rightarrow\ \kappa(\rho,\zeta)[\!]\ \zeta()$

$\mathcal{M}[\![\texttt{(cons}\ \varphi\ \varphi'\texttt{)}]\!] = \lambda\varepsilon\rho\mu\kappa\zeta.\ \varepsilon \in \mathbf{Pair} \rightarrow\ \mathcal{M}[\![\varphi]\!](\mathit{left}(\varepsilon),\rho,\mu,\lambda\rho'\zeta'.\mathcal{M}[\![\varphi']\!](\mathit{right}(\varepsilon),\rho',\mu,\kappa,\zeta'),\zeta)[\!]\ \zeta()$

$\mathcal{M}[\![\texttt{(or}\ \varphi\ \varphi'\texttt{)}]\!] =$
$\lambda\varepsilon\rho\mu\kappa\zeta.\mathcal{M}[\![\varphi]\!](\varepsilon,\rho,\mu,\kappa,\lambda().\mathcal{M}[\![\varphi']\!](\varepsilon,\rho,\mu,\kappa,\zeta))$

$\mathcal{M}[\![\texttt{(and}\ \varphi\ \varphi'\texttt{)}]\!] = \lambda\varepsilon\rho\mu\kappa\zeta.\mathcal{M}[\![\varphi]\!](\varepsilon,\rho,\mu,\lambda\rho'\zeta'.\mathcal{M}[\![\varphi']\!](\varepsilon,\rho',\mu,\kappa,\zeta'),\zeta)$

$\mathcal{M}[\![\texttt{(not}\ \varphi\texttt{)}]\!] = \lambda\varepsilon\rho\mu\kappa\zeta.\mathcal{M}[\![\varphi]\!](\varepsilon,\rho,\mu,\lambda\rho'\zeta'.\zeta(),\lambda().\kappa(\rho,\zeta))$

$\mathcal{M}[\![\texttt{(tree}\ \nu\ \varphi\ \varphi'\texttt{)}]\!] =$
$\lambda\varepsilon\rho\mu\kappa\zeta.\ \mathit{try}(\varepsilon,\rho,\kappa,\zeta)$
$\qquad\mathbf{whererec}\ \mathit{try} = \lambda\varepsilon'\rho'\kappa'\zeta'.\mathcal{M}[\![\varphi']\!](\varepsilon',\rho',\mu,\kappa',\lambda().\mathcal{M}[\![\varphi]\!](\varepsilon',\rho',\mu[\nu \rightarrow \mathit{try}],\kappa',\zeta'))$

$\mathcal{M}[\![\texttt{(hole}\ \nu\texttt{)}]\!] = \lambda\varepsilon\rho\mu\kappa\zeta.\mu(\nu)(\varepsilon,\rho,\kappa,\zeta)$

Table 2: Semantics of basic patterns

2 Partial evaluation

Partial evaluation transforms programs with incomplete input data. Let p be a source program written in language L with say, two arguments s and d, suppose moreover s to be known (static). A partial evaluator written in L, traditionally named *mix* after [JSS85], specializes program p (written in L) with respect to s into a residual program p_s (still written in L) equivalent to $\lambda d.p(s,d)$. The transformation does not have to be a simple curryfication on the static argument but is expected to propagate constants, reduce redexes etc. The specialized program is potentially more efficient than the original program since all the computations that depend on the static data have been performed. The partial evaluator, *mix*, satisfies the following property, expressed with the usual notations of partial evaluation:

$$L\ p(s,d) = L\ p_s(d) \text{ where } p_s = L\ \mathit{mix}(p,s)$$

Suppose now *int* to be an interpreter of some language, *int* takes a program p, some data d and computes the value of p with input d. $\mathit{mix}(\mathit{int},p)$ computes int_p which is the interpreter *int* specialized on p. The program int_p is similar to a compiled version of p as shown by:

$$L\ \mathit{int}(p,d) = L\ \mathit{int}_p(d) \text{ where } L\ \mathit{int}_p = L\ \mathit{mix}(\mathit{int},p)$$

The compiler itself from p to int_p is the result of $\mathit{mix}(\mathit{mix},\mathit{int})$ [Fut71]:

$$L\ \mathit{mix}(\mathit{mix},\mathit{int})(p)(d) = L\ \mathit{mix}_{\mathit{int}}(p)(d) = L\ \mathit{mix}(\mathit{int},p)(d) = L\ \mathit{int}_p(d) = L\ \mathit{int}(p,d)$$

Specializing *mix* on *mix* itself and the pattern matcher interpreter allows to automatically generate a pattern compiler. The pattern matcher interpreter of section 1 is approximately 1K pairs long, the corresponding generated compiler is 10K pairs long. The improved pattern matcher with intelligent backtrack below presented is 4K long and the corresponding compiler is 24K long. We use Similix [BD90] for all the experiments reported here.

$$
\begin{array}{rlcl}
\kappa \in & \textbf{Cont} & = & \textbf{Env} \times \textbf{Desc} \times \textbf{Alt} \rightarrow \textbf{Ans} \\
\zeta \in & \textbf{Alt} & = & \textbf{Desc} \rightarrow \textbf{Ans} \\
\mu \in & \textbf{Rep} & = & \textbf{Id} \rightarrow \textbf{Val} \times \textbf{Env} \times \textbf{Desc} \times \textbf{Cont} \times \textbf{Alt} \rightarrow \textbf{Ans} \\
\Phi \in & \textbf{Desc} & = & \textbf{Pat}
\end{array}
$$

$$
\mathcal{M} : \textbf{Pat} \rightarrow \textbf{Val} \times \textbf{Env} \times \textbf{Desc} \times \textbf{Rep} \times \textbf{Cont} \times \textbf{Alt} \rightarrow \textbf{Ans}
$$

Table 3: New domains for intelligent backtrack

3 Intelligent backtrack

Blind backtrack forgets all informations acquired on the datum between the last point of choice (an `or` or `tree` pattern) and the failure point. With respect to the basic set of patterns presented in section 1, failures can only occur in the `quote`, `cons` and (if bound) `var` patterns when a datum is not the expected value or not a dotted pair. Instead of just calling the failure continuation without information, we can pass it the description of the already inspected part of the datum. This description is usually partial since failures can be detected before the datum is completely scanned. Since patterns are a means to express sets of values with a similar shape, the best possible representation for the description is to be itself a pattern ! A datum differing from `(quote` ε`)` would be described by `(not (quote` ε`))`, a non dotted pair datum by `(not (cons (any) (any)))` and a datum not matched by a pattern variable ν by `(not (var` ν`))`.

The pattern matcher now has two supplementary arguments: a description, and an environment mapping pattern variables to descriptions. At any step, it compares a datum against a pattern under the control of a description, a success and a failure continuations. The datum is, by construction, always matched by the description, this invariant is maintained by the pattern matcher. Of course the initial description is the most undefined and uninformative pattern: `(any)` but more precise initial descriptions can also be imagined, see section 7 later. Before comparing the datum and the pattern, the description and the pattern are checked for compatibility. Three interesting cases can be recognized:

- the description and the pattern are disjoint i.e. no datum can satisfy simultaneously both of them. Failure is then guaranteed *without* looking at the datum.
- the description is contained in the pattern i.e. the datum is necessarily accepted by the pattern. Success is then guaranteed *without* looking to the datum.
- the description and the pattern have a common intersection so it is possible for the datum to be accepted by the pattern. It is then mandatory to effectively compare the datum and the pattern.

We therefore "reduce" the problem of pattern matching to the (harder) problem of patterns compatibility. Nevertheless, we can conservatively approximate the compatibility algorithm and still offers some improvements in the matching process. We will return to this point later.

The definition of the pattern matcher with intelligent backtrack appears in table 4. Semantics of revised domains are shown on table 3.

Except for the patterns `cons`, `var` and `quote` that are responsible for updating the description when tests are performed on the datum, the other patterns just make descriptions circulate. The `quote` pattern updates the incoming description Φ to be equal to `(quote` ε`)` if success or to `(and` Φ `(not (quote` ε`)))` if failure. This new description is returned to the appropriate continuation.

The `(var` ν`)` pattern updates the description with either `(var` ν`)` or `(not (var` ν`))`. The bound value can be found in the environment.

The `cons` pattern is slightly more complex. First it checks the incoming description Φ whether it *always* describe pairs or not, this is achieved by the *PAIR?* function. If the description is too uninformative then the datum has to be checked whether it is actually a dotted pair or not. Depending on the type of the datum, the description is updated to become `(and` Φ `(cons (any) (any)))` or `(and` Φ `(not (cons (any) (any))))`. If not already failing, matching continues and compares the left branch of the datum against the left branch of the pattern, an appropriate description for this left branch is computed from Φ with *LEFT*. After this match, the success or the failure continuation is invoked with an updated description Φ' for the left branch, this description is memorized until the end of the match where it will be combined with the description of the *RIGHT* branch, Φ'', to become the updated description of the whole dotted pair.

Obviously if *PAIR?* always return false and if *LEFT* and *RIGHT* always return the uninformative `(any)` pattern then the new pattern matcher is similar to the naïve one. It is nevertheless easy to give more accurate definitions for these three functions.

4 Application to strings

As a first application, let us choose string matching. Since we are in a Lisp setting, we encode strings as lists of symbols representing characters. Suppose we are looking for the substring `BABAR`. This problem is represented by the following pattern:

```
(tree skip (cons (any) (hole skip))
           (cons (quote B)
            (cons (quote A)
             (cons (quote B)
              (cons (quote A)
               (cons (quote R) (any)) ) ) ) ) )
```

The corresponding compiled code is:

```
(DEFINE (MATCH-0 E_0) (MATCH-4 E_0))

(DEFINE (MATCH-4 Z_1) ;see state 0 of figure 1
  (IF (PAIR? Z_1)
      (IF (EQUAL? (CAR Z_1) 'B)
          (MATCH-12 (CDR Z_1) Z_1)
          (MATCH-4 (CDR Z_1)))
      (FAIL)))

(DEFINE (MATCH-12 E_0 Z_1) ;see states 1 and 2 of figure 1
  (IF (PAIR? E_0)
      (IF (EQUAL? (CAR E_0) 'A)
          (LET ((E_3 (CDR E_0)))
            (IF (PAIR? E_3)
                (IF (EQUAL? (CAR E_3) 'B)
                    (MATCH-28 (CDR E_3) Z_1)
                    (MATCH-4 (CDDDR Z_1)))
                (FAIL)))
          (LET ((E_17 (CDR Z_1)))
            (MATCH-146 (CAR E_17) E_17)))
      (FAIL)))

(DEFINE (MATCH-28 E_0 Z_1) ;see states 3 and 4 of figure 1
  (IF (PAIR? E_0)
      (IF (EQUAL? (CAR E_0) 'A)
          (LET ((E_3 (CDR E_0)))
            (IF (PAIR? E_3)
                (IF (EQUAL? (CAR E_3) 'R)
                    (SUCCESS)
                    (IF (EQUAL? (CADDDDR Z_1) 'B)
                        (MATCH-28 (CDDDDDR Z_1) (CDDR Z_1))
                        (MATCH-4 (CDDDDDR Z_1))))
                (FAIL)))
          (LET ((E_35 (CDDDR Z_1)))
            (MATCH-146 (CAR E_35) E_35)))
      (FAIL)))
```

```
(DEFINE (MATCH-146 E_0 Z_3) ;see state 0 of figure 1
  (IF (EQUAL? E_0 'B)
      (MATCH-12 (CDR Z_3) Z_3)
      (MATCH-4 (CDR Z_3))))
```

This program comprises four functions nearly corresponding to the states of the automaton of figure 1. Some states are gathered into a single function but it is not a general rule since, for instance, state 0 is split into the two functions `MATCH-4` and `MATCH-146`, the last one knows that the list is not finished whereas the first has to check it. One can recognize that this automaton is similar to the automaton generated by the Knuth-Morris-Pratt (KMP) algorithm [KMP77].

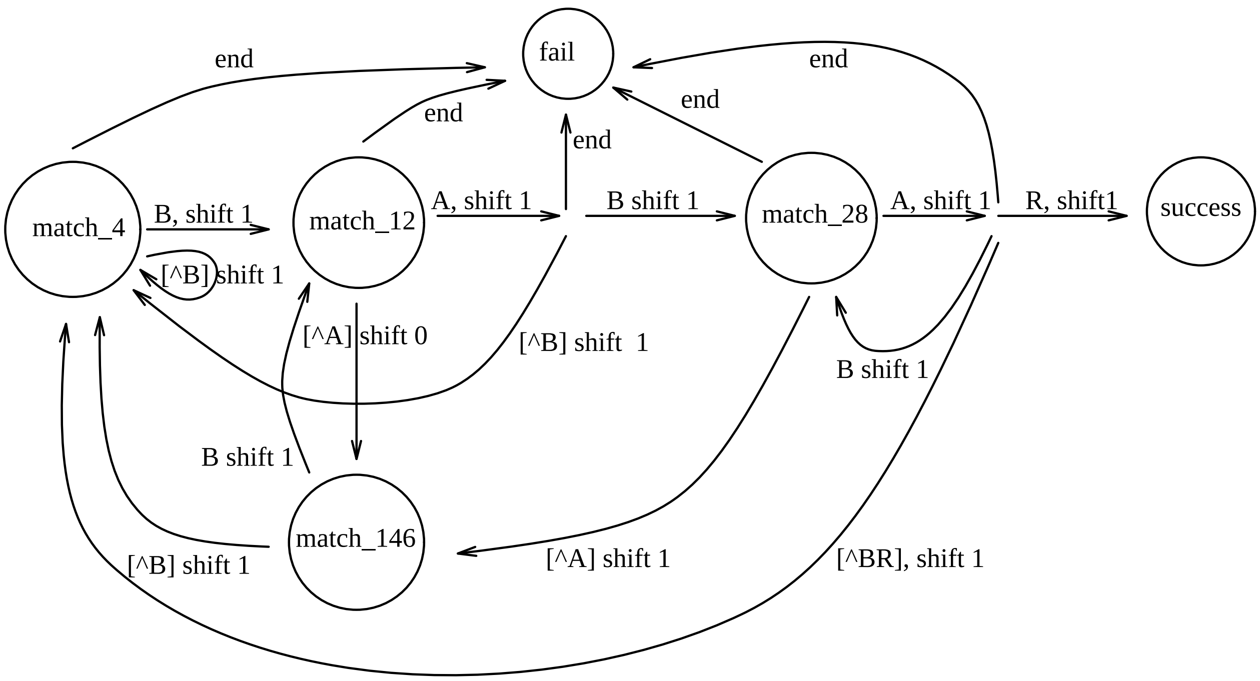

Figure 1: The `BABAR` automaton[2]

The technique is nonetheless slightly different since intelligent backtrack ignores completely the input letters and is only based on how to deal with descriptions corresponding to rejected data. Suppose for instance that the segment `B A B A` is already recognized. If the following letter is not an `R` then the description of the rejected datum is:

```
(cons (quote B)
 (cons (quote A)
  (cons (quote B)
   (cons (quote A)
    (cons (not (quote R))
          (any) ) ) ) ) )
```

The failure continuation is then invoked on this description. The most recent point of choice is given by the `skip` label of the enclosing `tree` pattern. This pattern skips the datum one position to the right and tries to resume matching with a new description shifted one place. This description `(cons (quote A) ...)` cannot fit the pattern which starts with a `B` so the failure continuation is called again on the new shifted

[2]The `end` symbol labels transitions occurring if the string is empty. `shift n` replaces the string by itself after erasure of its n first letters. `[^`$xy\ldots$`]` labels a transition triggered by a letter different from $xy\ldots$

description: `(cons (quote B) (cons (quote A) (cons (not (quote R)) (any))))`. This eliminates all tests on `B` and `A` and allows to directly return to state 2 where the input letter will be tested again to induce a new transition. In state 2, the input letter is only described by `(not (quote R))` which is compatible of both letters `A` and `B`.

This KMP-like behavior is not restricted to `cons` and `quote` patterns but can also be observed with the `var` pattern. The following pattern looks for lists containing a contiguous sequence of at least three equal terms:

```
(tree skip (cons (any) (hole skip))
           (cons (var x)
                 (cons (var x)
                       (cons (var x) (any)) ) ) )
```

Compared to the datum `A A B`, a failure on `B` would induce a shift of two places to remove the two `A`s.

The above technique naturally extends to the search of a keyword among a set of keywords and allows to rediscover the Aho-Corasick behavior [Aho90]. This is a good quality since these algorithms are very difficult to write by hand and so is the general algorithm. A related example appears at the end of section 5.

5 The influence of `cons`

The semantics of the `cons` pattern follows a left to right order. This section considers a variant named `xcons` with the reverse order. The `(xcons` ϕ_1 ϕ_2`)` pattern first checks if the datum is a pair, then tries to match the right branch of the datum against ϕ_2 then the left branch against ϕ_1. This simple modification allows us to exhibit a Boyer-Moore-like behavior (BM) [BM77]. Consider, for instance, the pattern:

```
(tree skip (xcons (any) (hole skip))
           (xcons (quote F) (xcons (quote O) (xcons (quote O) (any)))) )
```

This pattern searches the `F O O` substring. The generated code is composed of three functions and roughly corresponds to figure 2.

```
(DEFINE (MATCH-0 E_0)
  (IF (PAIR? E_0)
      (LET ((E_1 (CDR E_0)))
        (IF (PAIR? E_1)
            (LET ((E_2 (CDR E_1)))
              (IF (PAIR? E_2)
                  (LET ((E_3 (CDR E_2)))
                    (IF (EQUAL? (CAR E_2) 'O)
                        (IF (EQUAL? (CAR E_1) 'O)
                            (IF (EQUAL? (CAR E_0) 'F)
                                (SUCCESS)
                                (MATCH-0 (CDDDR E_0)))
                            (LET* ((E_10 (CDR E_0)) (E_11 (CDR E_10)))
                              (MATCH-156 (CDR E_11) E_10)))
                        (LET ((E_14 (CDDR E_0)))
                          (MATCH-216 (CDR E_14) E_14))))
                  (FAIL)))
            (FAIL)))
      (FAIL)))
(DEFINE (MATCH-216 E_0 Z_4)
  (IF (PAIR? E_0)
      (LET ((E_5 (CDR E_0)))
        (IF (PAIR? E_5)
            (LET ((E_6 (CDR E_5)))
             (IF (EQUAL? (CAR E_5) 'O)
                  (IF (EQUAL? (CAR E_0) 'O)
                      (IF (EQUAL? (CAR Z_4) 'F)
                          (SUCCESS)
```

```
                         (MATCH-0 (CDDDR Z_4))
                       (LET* ((E_14 (CDR Z_4)) (E_15 (CDR E_14)))
                         (MATCH-156 (CDR E_15) E_14))))
                   (LET ((E_18 (CDDR Z_4)))
                     (MATCH-216 (CDR E_18) E_18))))
             (FAIL)))
       (FAIL)))
 (DEFINE (MATCH-156 E_0 Z_6)
   (IF (PAIR? E_0)
       (LET ((E_7 (CDR E_0)))
         (IF (EQUAL? (CAR E_0) 'O)
             (IF (EQUAL? (CAR Z_6) 'F)
                 (SUCCESS)
                 (LET* ((E_13 (CDR (CDR Z_6))) (E_14 (CDR E_13)))
                   (MATCH-0 E_14)))
             (LET ((E_16 (CDDR Z_6)))
               (MATCH-216 (CDR E_16) E_16))))
       (FAIL)))
```

Letters are tested from the end as in BM. In a sense, the generated code is better than BM and KMP since the first only generates states from suffixes whereas KMP generates states from prefixes. Here, partial evaluation leads to the extra state `-O-` where only the second letter is known. This allows to combine the largest jumps and no loss of informations. If the alphabet is large, these improvements are probably worthless but they gain interest for small alphabets for example when matching sequences of nucleotides.

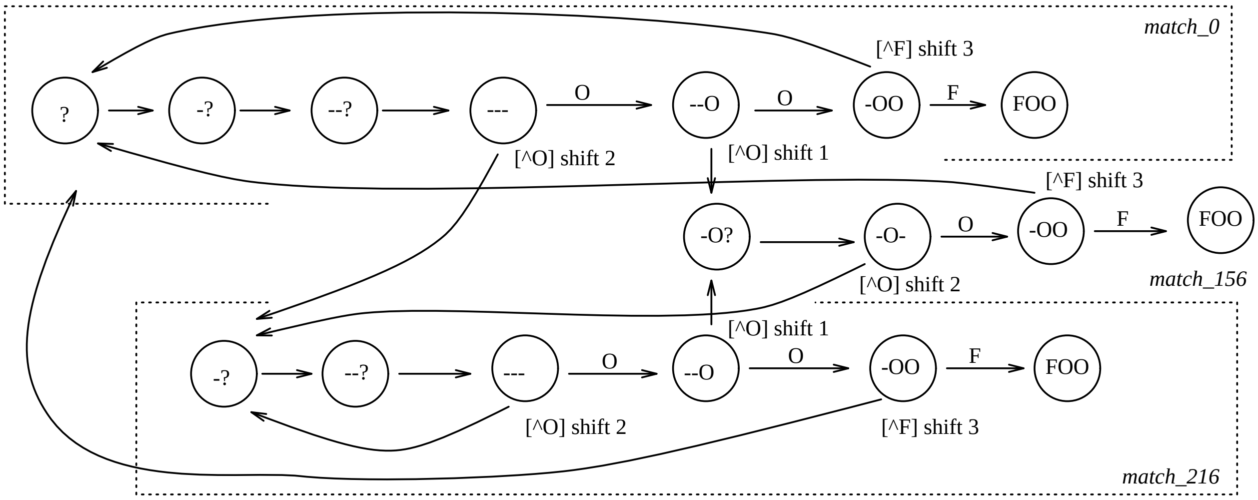

Figure 2: The `FOO` automaton[3]

It is also possible to mix `cons` and `xcons` patterns as partially done in the previous example. Let us, for example, search for one keyword among the set of keywords `B M` or `K M P`, moreover the search must be à la Boyer-Moore: this is known as the Commentz-Walter algorithm [Aho90]. The pattern is:

```
(tree skip (cons (any) (hole skip))
           (or (xcons (quote B) (xcons (quote M) (any)))
               (xcons (quote K) (xcons (quote M) (xcons (quote P) (any)))) ) )
```

The generated code contains only two functions which check if `M` appears with a preceding `B` otherwise `M` must be surrounded with `P` and `K`.

[3] States are labelled with the shape of the known prefix of the string to match. `?` means that the existence of a letter has to be checked while `-` means that the corresponding letter exists but is yet unknown. Success is represented by the `FOO` state.

```
(DEFINE (MATCH-0 E_0) (MATCH-6 E_0))
(DEFINE  (MATCH-6 Z_2)
  (IF (PAIR? Z_2)
      (LET ((E_3 (CDR Z_2)))
        (IF (PAIR? E_3)
            (LET ((E_4 (CDR E_3)))
              (IF (EQUAL? (CAR E_3) 'M)
                  (IF (EQUAL? (CAR Z_2) 'B)
                      (SUCCESS)
                      (LET* ((E_7 (CDR Z_2)) (E_8 (CDR E_7)))
                        (IF (PAIR? E_8)
                            (LET ((E_9 (CDR E_8)))
                              (IF (EQUAL? (CAR E_8) 'P)
                                  (LET ((E_11 (CAR E_7)))
                                    (IF (EQUAL? (CAR Z_2) 'K)
                                        (SUCCESS)
                                        (LET ((E_15 (CDR (CDR (CDR Z_2)))))
                                          (MATCH-6 E_15))))
                                  (LET ((E_17 (CDR (CDR Z_2))))
                                    (MATCH-159 (CDR E_17) E_17))))
                            (FAIL))))
                  (LET ((E_21 (CDR Z_2)))
                    (MATCH-159 (CDR E_21) E_21))))
            (FAIL)))
      (FAIL)))
(DEFINE (MATCH-159 E_0 Z_5)
  (IF (PAIR? E_0)
      (LET ((E_6 (CDR E_0)))
        (IF (EQUAL? (CAR E_0) 'M)
            (IF (EQUAL? (CAR Z_5) 'B)
                (SUCCESS)
                (LET* ((E_9 (CDR Z_5)) (E_10 (CDR E_9)))
                  (IF (PAIR? E_10)
                      (LET ((E_11 (CDR E_10)))
                        (IF (EQUAL? (CAR E_10) 'P)
                            (LET ((E_13 (CAR E_9)))
                              (IF (EQUAL? (CAR Z_5) 'K)
                                  (SUCCESS)
                                  (LET ((E_17 (CDR (CDR (CDR Z_5)))))
                                    (MATCH-6 E_17))))
                            (LET ((E_19 (CDR (CDR Z_5))))
                              (MATCH-159 (CDR E_19) E_19))))
                      (FAIL))))
            (LET ((E_23 (CDR Z_5))) (MATCH-159 (CDR E_23) E_23))))
      (FAIL)))
```

One interest of our technique is that the writer of a precise pattern does not have to master these very clever but complex algorithms and still obtain a good generated program. On the other hand, the generated code tends to be unpredictably voluminous.

6 Application to trees

Our pattern language has the power of describing trees so the same improvements can be observed on trees. The following example of tree-matching is used by [HO82]:

```
(tree aabvv (cons (hole aabvv) (hole aabvv))
      (cons (quote a)
            (cons (cons (quote a) (cons (quote b) (any)))
```

```
                    (any) ) ) )
```

Once again the generated code is good (but voluminous and not shown here) and behaves as the algorithm described in [HO82].

7 The influence of the initial description

The initial description appears free in our algorithm. It can be tailored to suit particular needs. Consider, for instance, the following initial description:

```
(tree only-F-and-O (cons (or (quote F) (quote O)) (hole only-F-and-O))
                    (quote ()) )
```

It describes a set of possible inputs: these must be regular lists containing only `F` and `O`. With this hypothesis, the program recognizing `F O O` is greatly simplified since a description as `(not (quote F))` is equivalent to `(quote O)`.

Appropriate initial descriptions allow to precisely characterize the set of input data. The classical `append` function by case analysis à la ML can be written as:

```
(append (cons (var head) (var tail)) (var list)) -> ...1
(append (quote ()) (var list))                   -> ...2
```

An initial description like `(tree List (cons (any) (hole List)) (quote ()))` would allow to specify that the input data is either the empty list or a dotted pair and nothing else. The generated equivalent code would look like `(if (pair?` ε`)` $\ldots_1$ $\ldots_2$`)`. Without the initial description the generated code would have been: `(if (pair?` ε`)` $\ldots_1$ `(if (equal?` ε `'())` $\ldots_2$ `(z.init) ) )`. This would have added an extraneous *otherwise* clause and an additional test.

8 Related Work

Pattern matching is one of the favorite fields of partial evaluation [CD89, Jør90, Dan91, Smi91]. It is often claimed that partial evaluation can "automatically" rediscover clever algorithms such as Knuth-Morris-Pratt. In fact partial evaluation, being a semantics-preserving transformation, is completely unable to rediscover anything that is not somehow already present in the algorithm to be partially evaluated. Starting from a naïve pattern matcher, a very slight modification suggesting that the translation of the pattern can be computed from the known prefixes of the matched input string allows, using partial evaluation to eliminate all redundant tests and obtain a program which is structurally equivalent to the automaton described in [KMP77]. Examples of KMP derivations appear in [CD89, Dan91, Smi91] as well as derivation of BM [Amt90, **??Danvy91e??**] ; a discussion on the varying naïvety of such input programs may be found in [Smi91].

Rather than trying to obtain known algorithms from more or less naïve pattern matchers, we tried the other way round i.e. we extend the pattern matcher to acquire the capability of intelligent backtrack. The algorithm is very general, does not require to extend the power of regular partial evaluator and can be applied to various problems with interesting results. There is no rediscovery, no eurekas but only the thorough application of a single principle which is "do not forget informations between failures and choice points".

Although simple, this principle turns out to have important applications. In the string matching domain, for instance, the KMP behavior was easy to obtain [Dan91, Smi91] since the letters of the prefix before the failure are known. With the BM behavior, these are completely unknown, only the structure of the failure is known and has to be dealt with. It is nevertheless possible to obtain BM behavior as exposed in —citeAmtoft90,Danvy91e.

A close work is [Jør90] from which we borrow the word *description*. Jørgensen used a combinator language with linear patterns *à la* ML, he gives it a semantics where every time a value is tested, the associate description is updated to memorize the result. This allows to eliminate redundant tests and to avoid surely failing matches, the technique generates very good code. Our work uses the pattern language itself to represent descriptions and it offers a richer set of patterns. We show how the semantics of constructors (`cons`, `xcons`) influence the behavior of the pattern matcher. We also provide an elegant way to introduce a priori information on data to be matched by means of the initial description. This allows to generate very good code even in an untyped language such as Lisp.

9 Conclusion

We present a pattern matching algorithm implementing an intelligent backtrack facility. Various definitions of the semantics of some pattern constructors make this algorithm look as if it offers many of the very clever ideas of the well-known Knuth-Morris-Pratt or Boyer-Moore algorithms. The main interests of our approach is

1. patterns are pretty easy to write
2. partial evaluation automatically generates compilers from definitional interpreters
3. the generated code is good.

On the other side, our algorithm being very general, it cannot discover clever data structures such as jump tables nor it can imagine *eurekas* such as Karp-Rabin method. The size of the generated code is usually large, sometimes exponential and, most of the time, unpredictable. To keep all informations for intelligent backtrack can be too expensive both at run-time or at partial-evaluation time. This can be partially alleviated by new definitions for *PAIR?*, *LEFT* and *RIGHT* functions to forget some informations. Some analyzes must then be done to discover which informations are important since many of them only provide very small optimizations. This is leaved for future research.

Acknowledgement

Thanks to Bernard Lang who suggested adding intelligent backtrack to our previous pattern compiler, to Philippe Codognet who gave us his "best of" papers on intelligent backtrack in Prolog, to Pierre Weis who knows a lot on pattern matching for ML, to Jesper Jørgensen who kindly suggests many improvements and refers us to the work of Amtoft.

Bibliography

[Aho90] Alfred Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 273–282. Elsevier Science Publishers, 1990.

[Amt90] Torben Amtoft. Transforming a naive pattern matcher into efficient pattern matchers. Unpublished manuscript from DAIMI, University of Aarhus DK-8000 Aarhus C, Denmark `tamtoft@daimi.aau.dk`, December 1990.

[BD90] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. Technical Report 90/04, DIKU, University of Copenhaguen, Denmark, 1990.

[BEJ88] D Bjørner, A Ershov, and N D Jones, editors. *Partial Evaluation and Mixed Computation*, Avernæs (Denmark), October 1987 1988. North Holland.

[BM77] R S Boyer and J S Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):62–72, 1977.

[CD89] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(1), September 1989.

[Dan91] Olivier Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37:315–322, 1991.

[Fut71] Y Futamura. Partial evalaution of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[HJ91] Paul Hudak and Neil D Jones, editors. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Sigplan Notices 26(9), New Haven (Connecticut USA), June 1991. Association of Computing Machinery.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[IEE91] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[Jør90] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming*, Workshops in Computing, pages 177–195, Glasgow (UK), August 1990. Springer-Verlag.

[JSS85] N D Jones, P Sestoft, and H Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In *Rewriting Techniques and Applications*, volume Lecture Notes in Computer Science 202, pages 124–140. Springer-Verlag, 1985.

[KMP77] D E Knuth, J H Morris, and V R Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[Que90] Christian Queinnec. Compilation of Non-Linear, Second Order Patterns on S-Expressions. In P Deransart and J Małuszyński, editors, *Lecture Notes in Computer Science 456*, pages 340–357, Linköping, August 1990. International Workshop PLILP '90 – Programming Language Implementation and Logic Programming, Springer-Verlag.

[Sch86] David A Schmidt. *Denotational Semantics, a Methodology for Language Development*. Allyn and Bacon, 1986.

[Smi91] Donald A Smith. Partial evaluation of pattern matching in constraint logic programming languages. In *PEPM '91 – Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71, Yale University (Connecticut USA), June 1991. ACM Press. Also SIGPLAN Notices 22(9), September 1991.

$$
\begin{aligned}
\mathcal{M}[\![\varphi]\!] = \lambda\varepsilon\rho\Phi\mu\kappa\zeta.\Phi\tilde{\subset}\varphi &\rightarrow \kappa(\rho,\Phi,\zeta'')\\
[\!]\ \Phi\tilde{\wedge}\varphi \neq \emptyset &\rightarrow \mathcal{M}'[\![\ \varphi]\!]\ (\varepsilon\rho\Phi\mu\kappa\zeta)\\
&[\!]\ \zeta(\Phi)
\end{aligned}
$$

$$
\mathcal{M}'[\![\texttt{(any)}]\!] = \lambda\varepsilon\rho\Phi\mu\kappa\zeta.\kappa(\rho,\Phi,\zeta)
$$

$$
\begin{aligned}
\mathcal{M}'[\![\texttt{(quote } \varepsilon\texttt{)}]\!] = \ \textbf{let}\ \Phi = \texttt{(quote } \varepsilon\texttt{)}\ \textbf{in}\ \lambda\varepsilon'\rho\Phi'\mu\kappa\zeta.\ \varepsilon' = \varepsilon \rightarrow\ & \kappa(\rho,\texttt{(and } \Phi'\ \Phi\texttt{)},\zeta)\\
[\!]\ & \zeta(\texttt{(and } \Phi'\ \texttt{(not } \Phi\texttt{))})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{M}'[\![\texttt{(var } \nu\texttt{)}]\!] = \lambda\varepsilon\rho\Phi\mu\kappa\zeta.\rho(\nu) = \mathit{unbound} \rightarrow\ & \kappa(\rho[\nu\rightarrow\varepsilon],\zeta)\\
[\!]\ \ \rho(\nu) = \varepsilon \rightarrow\ & \kappa(\rho,\Phi,\zeta)\\
[\!]\ & \zeta()
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{M}'[\![\texttt{(cons } \varphi\ \varphi'\texttt{)}]\!] =\\
&\quad \textbf{let}\ \Phi = \texttt{(cons } \varphi\ \varphi'\texttt{)}\\
&\quad \textbf{and}\ \Phi' = \texttt{(cons (any) (any))}\\
&\quad \textbf{in}\ \lambda\varepsilon\rho\Phi''\mu\kappa\zeta.\ \textbf{let}\ \theta = \lambda().\mathcal{M}[\![\varphi]\!]\ (\mathit{left}(\varepsilon),\rho,\mathrm{LEFT}(\texttt{(and } \Phi''\ \Phi'\texttt{)}),\mu,\\
&\qquad\qquad\qquad \lambda\rho'\Phi'''\zeta'.\mathcal{M}[\![\varphi']\!](\ \mathit{right}(\varepsilon),\rho',\mathrm{RIGHT}(\texttt{(and } \Phi''\ \Phi'\texttt{)}),\mu,\\
&\qquad\qquad\qquad\qquad \lambda\rho''\Phi''''\zeta''.\kappa(\rho'',\texttt{(cons } \Phi'''\ \Phi''''\texttt{)},\zeta''),\\
&\qquad\qquad\qquad\qquad \lambda\Phi''''.\zeta'(\texttt{(cons } \Phi'''\ \Phi''''\texttt{)})),\\
&\qquad\qquad\qquad \lambda\Phi'''.\zeta(\texttt{(and } \Phi''\ \texttt{(cons } \Phi'''\ \texttt{(any))))})\\
&\qquad\quad \textbf{in}\ \mathrm{PAIR?}(\Phi'')\ \rightarrow \theta()\\
&\qquad\qquad\qquad [\!]\ \varepsilon \in \mathbf{Pair} \rightarrow\ \theta()\\
&\qquad\qquad\qquad\qquad [\!]\ \zeta(\texttt{(and } \Phi''\ \texttt{(not } \Phi'\texttt{)))}
\end{aligned}
$$

$$
\mathcal{M}'[\![\texttt{(or } \varphi\ \varphi'\texttt{)}]\!] = \lambda\varepsilon\rho\Phi\mu\kappa\zeta.\mathcal{M}[\![\varphi]\!](\varepsilon,\rho,\Phi,\mu,\kappa,\lambda\Phi'.\mathcal{M}[\![\varphi']\!](\varepsilon,\rho,\Phi',\mu,\kappa,\zeta))
$$

$$
\mathcal{M}'[\![\texttt{(and } \varphi\ \varphi'\texttt{)}]\!] = \lambda\varepsilon\rho\Phi\mu\kappa\zeta.\mathcal{M}[\![\varphi]\!](\varepsilon,\rho,\Phi,\mu,\lambda\rho'\Phi'\zeta'.\mathcal{M}[\![\varphi']\!](\varepsilon,\rho',\Phi',\mu,\kappa,\zeta'),\zeta)
$$

$$
\mathcal{M}'[\![\texttt{(not } \varphi\texttt{)}]\!] = \lambda\varepsilon\rho\Phi\mu\kappa\zeta.\mathcal{M}[\![\varphi]\!](\varepsilon,\rho,\Phi,\mu,\lambda\rho'\Phi'\zeta'.\zeta(\Phi'),\lambda\Phi'.\kappa(\rho,\Phi',\zeta))
$$

$$
\begin{aligned}
&\mathcal{M}'[\![\texttt{(tree } \nu\ \varphi\ \varphi'\texttt{)}]\!] =\\
&\lambda\varepsilon\rho\Phi\mu\kappa\zeta.\mathit{try}(\varepsilon,\rho,\Phi,\kappa,\zeta)\\
&\ \textbf{whererec}\ \mathit{try} = \lambda\varepsilon'\rho'\Phi'\kappa'\zeta'.\mathcal{M}[\![\varphi']\!](\ \varepsilon',\rho',\Phi',\mu,\lambda\rho''\Phi''\zeta''.\kappa'(\rho'',\texttt{(and } \Phi''\ \varphi'\texttt{)},\zeta''),\\
&\qquad\qquad \lambda\Phi''.\mathcal{M}[\![\varphi]\!](\ \varepsilon',\rho,\Phi'',\mu[\nu\rightarrow \mathit{try}],\\
&\qquad\qquad\qquad \lambda\rho''\Phi'''\zeta''.\kappa'(\rho'',\texttt{(and } \Phi'''\ \varphi\texttt{)},\zeta''),\zeta'))
\end{aligned}
$$

$$
\mathcal{M}'[\![\texttt{(hole } \nu\texttt{)}]\!] = \lambda\varepsilon\rho\Phi\mu\kappa\zeta.\mu(\nu)(\varepsilon,\rho,\Phi,\kappa,\zeta)
$$

$$
\begin{aligned}
\text{Conventions:}\quad & \varphi\tilde{\subset}\varphi' \Rightarrow \varphi \subset \varphi'\\
& \varphi\tilde{\wedge}\varphi' = \emptyset \Rightarrow \varphi \wedge \varphi' = \emptyset
\end{aligned}
$$

Table 4: Pattern matcher with intelligent backtrack