Compilation of non-linear, second order patterns on S-expressions

PLILP '90, LNCS 456, pp 340–357.

Christian Queinnec*
École Polytechnique — INRIA

Abstract

Pattern matching is a key concept for rule-based expert systems. Simple pattern interpreters appear in nearly every book on Lisp. Pattern matching is also a useful tool for case analyses as provided by functional languages such as ML or Miranda^{TM}. These two uses are somewhat different since functional languages emphasize a discrimination based on types while Lisp, or Plasma, make use of S-expressions or segments of S-expressions within patterns. The paper presents an intermediate language for patterns and its denotational semantics. This reduced language is powerful enough to express boolean composition of patterns as well as segment handling and unbounded pattern repetition. A compiler is then defined which translates patterns into functional code. We discuss some compilation variant as well as the integration of the pattern sub-language into Lisp. These capabilities make pattern matching an useful and efficient tool for a wide class of applicative languages and allow to incorporate into patterns more knowledge about the form or *gestalt* of the data to be matched.

Pattern matching is gaining more and more audience within functional languages. Pattern matching is used to define functions, or just expressions, that perform case analyses. First introduced in NPL [Burstall 69] and SASL [Turner 76], then in ML [Weis 89] or Haskell [Hudak & Wadler 90] `case` construct is an archetypic example of the use of pattern matching within functional languages. Pattern matching is also a key concept in rule-based expert systems and has long been known in the Lisp community. Most text-books on Lisp [Abelson & Sussman 85, Kessler 88, Queinnec 84, Winston 88] contain the description of simple pattern interpreters. The first pattern compiler was written by Masinter [Teitelman 78] and offered since 1972 as part of the InterLisp programming environment.

Pattern matching compares a *datum* and a *pattern*. Two effects are simultaneously involved:

- check the aspect of the datum,
- destructure this very datum and extract some of its components.

The first effect yields a boolean result indicating if the datum matches the pattern i.e.there exists an *environment* binding *pattern variables* to data that makes the pattern, with pattern variables substituted by their value in this very environment, equal to the original datum. The second effect may be considered as a by-product since the destructuration is in fact the precise environment binding pattern variables to parts of the original datum and constructed as part of the matching process.

The omnipresent list structure of Lisp pushed it to emphasize the overall *form*: patterns describe sets of S-expressions and some notations exist to specify conveniently these sets. For instance `?`x is a pattern variable while `?-` represent a *wild card* matching anything with success. A sequence of data can be bound by `??`x, while `??-` represent a sequence of wild cards. Sets of S-expressions can be easily specified by patterns,

*Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex – France, (`queinnec@poly.polytechnique.fr`)

for example `(?x ??y ?x)` matches all lists that begin and end with a similar term[1]: this term and the inner part of the list are extracted under the names `x` and `y`.

On the other hand, pattern matching in ML is just a discrimination on the type. Repeated variables are often avoided in functional languages since they hurt the semantics of lazyness [Peyton Jones 86, page 65]. Segment patterns which may introduce multiple solutions[2] are not used at all while they might appear in ZF generators of Miranda.

Let us give some examples of these two cultures. The pattern `(??- ?x ??- ?x ??-)` matches every datum which is a list containing at least one repeated element. `?x` is the pattern variable identifying the duplicated element while `??-` represents a sequence of uninteresting elements. Consider the function `member` which checks if its first argument appears in the list which is its second argument. If written in ML (with our notation for pattern variables), `member` will typically look like:

```
member ?x ()        -> false
member ?x (?z . ?y) -> x = z or member ?x ?y
```

conversely in a Plasma-like [Hewitt & Smith 75] style, `member` will look like:

```
member ?x (??- ?x ??-) -> true
member ?x ?-           -> false
```

One difference is the use of `??-` involving an arbitrary number of dotted pairs between the head of the list and the element looked for. This facility does not exist in current ML. Conversely the ML style involves non overlapping patterns with different types while Lisp takes for granted the sequentiality of the two clauses defining `member`.

On the other side, a pattern such as `((*check number?) ...)`[3] which matches any (possibly empty) list of numbers is costly in Lisp since it requires an examination of the type of every element of the list. The same effect is for free in ML since the pattern may be statically type-checked[4] and recognized to only match lists of numbers. This verification is done at compile-time and thus does not involve any run-time penalty.

Our goal is to allow the use of lists (or, more generally, linked structure) within patterns as well as to give a precise semantics for conjunction, disjunction and negation of patterns, element or segment variable binding or reference. We first present the pattern language in section 1. Its denotational semantics appears in section 2. The relation between this compilation and the underlying language is analysed in section 3. The compilation of these patterns into a pure functional subset of Scheme is covered in section 4. Other compilation schemes and examples of code generation produced by a full blown pattern compiler are given in section 5 and 6. User interface is also dealt with and mostly the notion of macro patterns. Related works are discussed in section 7.

1 The Pattern Language

Our pattern language is made of a small number of primitive patterns or pattern schemes that can be recursively composed. There are 13 of them and they are named after their use in Lisp. Patterns can have components such as labels, values or other patterns; we thus adopt a parenthetized syntax. This pattern language was devised with two goals — to be small and expressive — while preserving a clean semantics and simple compilation schemata. The above examples of patterns such as `(??- ?x ??- ?x ??-)` do not belong to the reduced language we now introduce. They are too high level and mix several effects we want to distinguish. Nevertheless we will continue to use these convenient although extended patterns and will indicate *en passant* how they are translated in terms of the reduced language. An alternate view is to consider the pattern language as an intermediate language following the RISC philosophy.

Each pattern is presented and informally described hereafter:

[1] The first use of `?x` binds the pattern variable `x` while the second is a reference to this very value.

[2] Consider for instance `(??- ?x ??-)` which can bind the pattern variable `x` to any term of the datum. The datum is only required to be a list.

[3] The three dots indicate the unbounded repetition of the previous term.

[4] Patterns can be directly typed *à la ML*. This allows to produce meaningful error messages in the domain of patterns rather than relatively to the equivalent generated code.

The two following patterns are the leafs of more complex patterns.

→ `(*sexp)` is the pattern which accepts any datum. We previously nicknamed it with `?-`.

→ `(*quote` *value*`)` only accepts a datum equal[5] to *value*.

Patterns can be composed into lists thanks to `*cons`.

→ `(*cons` *pattern1 pattern2*`)` accepts any list which head is acepted by *pattern1* and which tail is accepted by *pattern2*.

It is possible to extract and store parts of the analysed data into pattern variables.

→ `(*setq` *name pattern*`)` accepts anything that *pattern* accepts then binds the pattern variable *name* to what is accepted. No previous binding of the same name must exist before: binding is done at most once. The notation `?x`, used for binding, is therefore an abbreviation of `(*setq x (*sexp))`.

→ `(*eval` *name*`)` only accepts a datum equal to the datum previously bound to the matching variable *name*. The pattern variable *name* must obviously be bound before. The notation `?x`, used for reference, is thus an abbreviation of `(*eval x)`.

Patterns can be composed thanks to boolean pattern composers.

→ `(*or` *pattern1 pattern2*`)` accepts anything that can be accepted by either *pattern1* or *pattern2*. In a `case` form, the different patterns are to be considered as implicitely `*or`-ed. These patterns may be cleverly compiled if they share common prefixes, see [Peyton Jones 86].

→ `(*and` *pattern1 pattern2*`)` accepts anything that *pattern1* and *pattern2* both accept. Moreover since *pattern1* is tried first, *pattern2* will be exercized in the environment resulting from *pattern1*. For instance in the pattern `(*and (??x ?-) (?- ??x))`, the second segment variable `??x` is used for reference since the first argument of `*and` will bind `x`. This pattern accepts any non empty list of which all terms are equal. Examples are `(bar)`, `(foo foo foo foo)`.

The `*and` pattern closely corresponds to the *as-pattern* which appears in ML or Haskell [Hudak & Wadler 90].

→ `(*not` *pattern*`)` accepts anything that is not accepted by *pattern*.

Segment variables may be bound or referenced.

→ `(*ssetq-append` *name pattern1 pattern2*`)` accepts any datum which first terms are accepted by *pattern1* while other terms are accepted by *pattern2*. Moreover, just after succeeding with *pattern1* and before *pattern2* is exercized, the pattern variable *name* is bound to what *pattern1* matched.

→ `(*eval-append` *name pattern*`)` accepts any datum which beginning is accepted by the value bound to *name* and which tail is matched by *pattern*.

→ `(*end-ssetq` *name*`)` marks the end of the *pattern1* argument of the associated `*ssetq-append` i.e.where to start to process *pattern2*.

A repetition is specified by a pattern that must be repeated some times before being followed a final pattern. There again, a special pattern marks the end of the repetition.

→ `(*times` *label pattern1 pattern2*`)` matches any datum which beginning is accepted by the possibly repeated pattern *pattern1* and which tail is matched by *pattern2*.

→ `(*end-times` *label*`)` marks the end of the repetition of the *pattern1* argument in the corresponding enclosing `*times`. The unmatched rest of the list is then submitted to *pattern2*.

[5] Other patterns can be introduced to test various equalities: for example, in the Lisp realm, `eq` or `equal`.

A small example will illustrate the use of all these previous patterns. The pattern `(??x ??x)` only accepts lists that are made of two repetitive segments. Some examples are `()`, `(bar bar)` or `(f o o f o o)`. The exact translation of `(??x ??x)` in terms of the reduced set of patterns is:

```
(*ssetq-append  x (*times loop (*cons (*sexp)
                                      (*end-times loop) )
                               (*end-ssetq x) )
                  (*eval-append x (*quote ())) )
```

The inner pattern `(*times ...)` is in fact equivalent to `(*end-ssetq x)` or `(*cons (*sexp) (*end-ssetq x))` or `(*cons (*sexp) (*cons (*sexp) (*end-ssetq x)))` etc. where `(*end-ssetq x)` means: bind now `x` and continue.

The reduced set of patterns is sufficient to express a wide variety of patterns. It is not convenient for direct use since it syntax is awkward and only suited for computers but it clearly separates effects such as binding, referencing, repetition, end of repetition ... One remaining problem we should not address in this paper is to devise a handy syntax for expressing patterns. Not specifying the external syntax allows us to embed our pattern intermediate language in a wide class of programming languages but for each embedding programming language, a natural syntax has to be devised. The main difficulty is, at least for us, that new, numerous although readable notations may introduce a syntactical cancer !

2 Pattern Semantics

The semantics of our pattern language is expressed in terms of denotational semantics [Stoy 77, Schmidt 86]. We consider important to give a precise formalization since many subtle questions might not be raised in an informal description. The domains are:

$\rho \in$	**Menv**	$=$	**Id** $\to$ **Val** $\cup \{$*unbound-pattern*$\}$
$\alpha \in$	**Seg**	$=$	**Id** $\to$ **Mcont**
$\mu \in$	**Rep**	$=$	**Id** $\to$ **Mcont**
$\kappa \in$	**Mcont**	$=$	**Val** $\times$ **Menv** $\times$ **Alt** $\to$ **MAnswer** /* Success */
$\zeta \in$	**Alt**	$=$	**Unit** $\to$ **MAnswer** /* Failure */
$\nu \in$	**Id**		The set of identifiers
$\varepsilon \in$	**Val**		The set of data to be matched
$\phi \in$	**Pattern**		The set of patterns

$\mathcal{M}$: **Pattern** $\to$ **Val** $\times$ **Menv** $\times$ **Seg** $\times$ **Rep** $\times$ **MCont** $\times$ **Alt** $\to$ **MAnswer**

The semantics of patterns is expressed by $\mathcal{M}$. $\mathcal{M}$ takes a datum ε to analyse, the environment ρ mapping pattern variables to data, the name α of the current segment (introduced by `*ssetq-append`) if any, the label μ of the current repetition (introduced by `*times`) if any, the success continuation κ and the failure continuation ζ of the matching process. **MAnswer** is the final answer that yields pattern matching, this domain will be clarified later. $\mathcal{M}$ may be seen as a "double continuation interpreter": every step of the matching process is associated with two continuations corresponding to the result of the step: success or failure. A consequence is that there exists an explicit sequentiality in the matching process in order to give a precise meaning to multiply-referenced variables and an order in the enumeration of multiple solutions. The present pattern semantics is totally left to right oriented.

Here is the semantics of the two fundamental patterns. Success is marked by $\kappa(\varepsilon, \rho, \zeta)$ while failure is invoked with $\zeta()$.

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

$$\mathcal{M}[\![\texttt{*quote}\ \varepsilon]\!] = \lambda\varepsilon'\rho\alpha\mu\kappa\zeta.\ \textbf{if}\ \varepsilon' = \varepsilon\ \textbf{then}\ \kappa(\varepsilon', \rho, \zeta)\ \textbf{else}\ \zeta()\ \textbf{endif}$$

Lists belong to the **Pair** domain and have a head ($\varepsilon\downarrow_1$) and a tail ($\varepsilon\S 1$). Observe that the head of the datum cannot contain the end of a segment variable nor the end of a repetition since this match is performed under α_{init} and μ_{init}. Both can only appear in the tail of the datum. Also note that tails are processed after heads, in the environment returned after matching heads.

$\mathcal{M}[\![$`*cons` $\phi_1\ \phi_2]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.$ **if** $\varepsilon \in$ **Pair**
then $\mathcal{M}[\![\phi_1]\!](\varepsilon\downarrow_1, \rho, \alpha_{init}, \mu_{init}, \lambda\varepsilon'\rho'\zeta'.\mathcal{M}[\![\phi_2]\!](\varepsilon\S 1, \rho', \alpha, \mu, \kappa, \zeta'), \zeta)$
else $\zeta()$ **endif**

$\alpha_{init}(\nu) = \lambda\varepsilon\rho\zeta.\mathit{wrong}($"`No current segment assignment for`"$, \nu)$

$\mu_{init}(\nu) = \lambda\varepsilon\rho\zeta.\mathit{wrong}($"`No current repetition named`"$, \nu)$

The semantics of conjunction and disjunction is simple but observe that the success continuation of the ϕ_1 argument of `*and` (which is to match the current data with ϕ_2) uses the environment resulting of ϕ_1. Once again the semantics sticks to the left-to-right order. The semantics of `*not` only consists to permute the two continuations. Note that the environment of success is lost if failure. Conversely choice points i.e. ζ' are lost if success.

$\mathcal{M}[\![$`*or` $\phi_1\ \phi_2]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{M}[\![\phi_1]\!](\varepsilon, \rho, \alpha, \mu, \kappa, \lambda().\mathcal{M}[\![\phi_2]\!](\varepsilon, \rho, \alpha, \mu, \kappa, \zeta))$

$\mathcal{M}[\![$`*and` $\phi_1\ \phi_2]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{M}[\![\phi_1]\!](\varepsilon, \rho, \alpha, \mu, \lambda\varepsilon'\rho'\zeta'.\mathcal{M}[\![\phi_2]\!](\varepsilon, \rho', \alpha, \mu, \kappa, \zeta'), \zeta)$

$\mathcal{M}[\![$`*not` $\phi]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{M}[\![\phi]\!](\varepsilon, \rho, \alpha, \mu, \lambda\varepsilon'\rho'\zeta'.\zeta(), \lambda().\kappa(\varepsilon, \rho, \zeta))$

Pattern variables are recorded in the environment ρ. Variables cannot be rebound nor they have a default value. The function *wrong* is not explicited and depends on the underlying language: *wrong* can report an error, raise an exception or return `nil` as usual in Lisp.

$\mathcal{M}[\![$`*setq` $\nu\ \phi]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{M}[\![\phi]\!](\varepsilon, \rho, \alpha, \mu, \lambda\varepsilon'\rho'\zeta'.$ **if** $\rho'(\nu) = \mathit{unbound\text{-}pattern}$
then $\kappa(\varepsilon, \rho'[\nu \to \varepsilon], \zeta')$
else $\mathit{wrong}($"`Cannot rebind pattern`"$, \nu)$
endif $, \zeta)$

$\mathcal{M}[\![$`*eval` $\nu]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.$ **if** $\rho(\nu) = \mathit{unbound\text{-}pattern}$
then $\mathit{wrong}($"`Unbound pattern`"$, \nu)$
else **if** $\rho(\nu) = \varepsilon$ **then** $\kappa(\varepsilon, \rho, \zeta)$ **else** $\zeta()$ **endif**
endif

$\rho_{init}(\nu) = \mathit{unbound\text{-}pattern}$

Segment variable binding and repetition must be correctly nested: α is therefore reset to α_{init} in the rule of `*ssetq-append`. The success continuation is also set to *wrong* since the effective assignment is only performed when encountering the associated `*end-ssetq`. This makes illegal a pattern such as `(*ssetq-append x (*sexp) ...)` where `(*end-ssetq x)` does not appear.

$\mathcal{M}[\![$`*ssetq-append` $\nu\ \phi_1\ \phi_2]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{M}[\![\phi_1]\!](\varepsilon, \rho, \alpha_{init}[\nu \to \lambda\varepsilon'\rho'\zeta'.$ **if** $\rho'(\nu) = \mathit{unbound\text{-}pattern}$
then $\mathcal{M}[\![\phi_2]\!](\varepsilon', \rho'[\nu \to \mathit{cut}(\varepsilon, \varepsilon')], \alpha, \mu, \kappa, \zeta')$
else $\mathit{wrong}($"`cannot rebind`"$, \nu)$
endif $],$
$\mu_{init}, \lambda\varepsilon'\rho'\zeta'.\mathit{wrong}($"`Ssetq not ended`"$, \phi_1), \zeta)$

$\mathcal{M}[\![$`*eval-append` $\nu\ \phi]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.$ **if** $\rho(\nu) = \mathit{unbound\text{-}pattern}$
then $\mathit{wrong}($"`Unbound segment`"$, \nu)$
else $\mathit{check}(\varepsilon, \rho(\nu), \lambda\varepsilon'.\mathcal{M}[\![\phi]\!](\varepsilon', \rho, \alpha, \mu, \kappa, \zeta), \zeta)$ **endif**

$\mathcal{M}[\![$`*end-ssetq` $\nu]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\alpha(\nu)(\varepsilon, \rho, \zeta)$

The library function *check* compares the current datum ε and the bound value $\rho(\nu)$, if the former begins by the latter then the success continuation is invoked $\psi(\varepsilon)$ otherwise failure $\zeta()$ is reported. The other library function *cut* returns the beginning of the list ε until finding the tail ε'.

$$\begin{aligned} check(\varepsilon, \varepsilon', \psi, \zeta) = &\ \mathbf{if}\ \varepsilon \in \mathbf{Pair}\ \wedge\ \varepsilon' \in \mathbf{Pair}\ \wedge\ \varepsilon\downarrow_1 = \varepsilon'\downarrow_1 \\ &\ \mathbf{then}\ check(\varepsilon \S 1, \varepsilon' \S 1, \psi, \zeta) \\ &\ \mathbf{else}\ \ \mathbf{if}\ \varepsilon' \in \mathbf{Null}\ \ \mathbf{then}\ \psi(\varepsilon)\ \mathbf{else}\ \zeta()\ \mathbf{endif} \\ &\ \mathbf{endif} \end{aligned}$$

$$cut(\varepsilon, \varepsilon') = \mathbf{if}\ \varepsilon = \varepsilon'\ \mathbf{then}\ <>\ \mathbf{else}\ <\varepsilon\downarrow_1> \S\, cut(\varepsilon \S 1, \varepsilon')\ \mathbf{endif}$$

Repetitions are handled in a similar way except that a local fix point is needed to produce the various solutions. μ_{init} contains this local iterator which is invoked with `*end-times` only.

$$\begin{aligned} &\mathcal{M}[\![\texttt{*times}\ \nu\ \phi_1\ \phi_2]\!] = \\ &\lambda\varepsilon\rho\alpha\mu\kappa\zeta.try(\varepsilon, \rho, \zeta) \\ &\qquad\quad \mathbf{whererec}\ try = \lambda\varepsilon'\rho'\zeta'. \\ &\qquad\qquad \mathcal{M}[\![\phi_2]\!](\varepsilon', \rho', \alpha, \mu, \kappa \\ &\qquad\qquad\qquad , \lambda().\mathcal{M}[\![\phi_1]\!](\varepsilon', \rho', \alpha_{init}, \mu_{init}[\nu \to try] \\ &\qquad\qquad\qquad\qquad\qquad , \lambda\varepsilon''\rho''\zeta''.wrong(\text{“}\texttt{Times not ended}\text{”}, \phi_1), \zeta')) \end{aligned}$$

$$\mathcal{M}[\![\texttt{*end-times}\ \nu]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mu(\nu)(\varepsilon, \rho, \zeta)$$

The pattern language above defined is very simple and can be extended if needed. For instance, it is convenient to add a more general pattern to match data that are not dotted pairs: `(*struct cons :car` ϕ_1 `:cdr` ϕ_2`)` will then be equivalent to `(*cons` ϕ_1 ϕ_2`)`[6]. This `*struct` pattern allows to iterate on any kind of linked structures. Again a convenient syntax is to be invented.

3 Integration within a Language

Pattern matching is a linguistic tool that must be grafted onto a language. We here choose a pure subset of Scheme [Rees & Clinger 86]. To shorten the paper we exclude assignment from the language and therefore we also exclude the **Store** domain from the denotation. Since multiple matches can be obtained, we keep a continuation based semantics to be able to express the order of evaluation.

$$\begin{array}{llll} \rho_L & \in & \mathbf{Env} & = \mathbf{Id} \to \mathbf{Val} \\ \kappa_L & \in & \mathbf{Cont} & = \mathbf{Val} \to \mathbf{Val} \\ \varphi & \in & \mathbf{Fun} & = \mathbf{Val}^* \times \mathbf{Cont} \to \mathbf{Answer} \\ \varepsilon & \in & \mathbf{Val} & = \mathbf{Fun} + \mathbf{Pair} + \ldots \\ \pi & \in & \mathbf{Program} & = \text{the set of programs} \\ \nu & \in & \mathbf{Id} & = \text{the set of identifiers} \\ \mathcal{E} & : & \multicolumn{2}{l}{\mathbf{Program} \to \mathbf{Env} \times \mathbf{Cont} \to \mathbf{Answer}} \\ \mathcal{E}^* & : & \multicolumn{2}{l}{\mathbf{Program}^* \to \mathbf{Env} \times \mathbf{Cont} \to \mathbf{Answer}} \end{array}$$

Some special forms are offered: `if`, `lambda`, `rec` and `match-all-lambda`. The semantics of the usual special forms is standard:

$$\mathcal{E}[\![\nu]\!] = \lambda\rho_L\kappa_L.\kappa_L(\rho_L(\nu))$$

$$\mathcal{E}[\![\texttt{rec}\ \nu\ \pi]\!] = \lambda\rho_L\kappa_L.\kappa_L(fix(\lambda\phi.\mathcal{E}[\![\pi]\!](\rho_L[\nu \to \phi], \lambda\varepsilon.\varepsilon)))$$

$$\begin{aligned} &\mathcal{E}[\![\texttt{if}\ \pi\ \pi'\ \pi'']\!] = \\ &\lambda\rho_L\kappa_L.\mathcal{E}[\![\pi]\!](\rho_L, \lambda\varepsilon.\ \mathbf{if}\ \varepsilon\ \mathbf{then}\ \mathcal{E}[\![\pi']\!](\rho_L, \kappa_L)\ \mathbf{else}\ \mathcal{E}[\![\pi'']\!](\rho_L, \kappa_L)\ \mathbf{endif}\ ) \end{aligned}$$

[6] Accessorily this new pattern allows to specify the matching order of the various subfields of the datum: `(*struct cons :cdr` ϕ_2 `:car` ϕ_1`)` is therefore different from `(*cons` ϕ_1 ϕ_2`)`. Moreover the first parameter can be allowed to be any known class name: the datum must then be an instance of that class or an instance of one of its subclasses.

$\mathcal{E}[\![\texttt{lambda}\ \nu^*\ \pi]\!] =$
$\lambda\rho_L\kappa_L.\kappa_L(\lambda\varepsilon^*\kappa_L'.\ \textbf{if}\ \#\nu^* = \#\varepsilon^*$
$\quad\textbf{then}\ \mathcal{E}[\![\pi]\!](\rho_L[\nu^* \xrightarrow{*} \varepsilon^*], \kappa_L')$
$\quad\textbf{else}\ \mathit{wrong}(\text{“}\texttt{Incorrect number of arguments}\text{”}, \nu^*)$
$\quad\textbf{endif}\ )$

$$\mathcal{E}[\![\texttt{begin}\ \pi_1\ \pi_2]\!] = \lambda\rho_L\kappa_L.\mathcal{E}[\![\pi_1]\!](\rho_L, \lambda\varepsilon.\mathcal{E}[\![\pi_2]\!](\rho_L, \kappa_L))$$

$$\mathcal{E}[\![\pi\ \pi^*]\!] = \lambda\rho_L\kappa_L.\mathcal{E}[\![\pi]\!](\rho_L, \lambda\varphi.\mathcal{E}^*[\![\pi^*]\!](\rho_L, \lambda\varepsilon^*.\varphi(\varepsilon^*, \kappa_L)))$$

$$\mathcal{E}^*[\![\ ]\!] = \lambda\rho_L\kappa_L.\kappa_L <>$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!] = \lambda\rho_L\kappa_L.\mathcal{E}[\![\pi]\!](\rho_L, \lambda\varepsilon.\mathcal{E}^*[\![\pi^*]\!](\rho_L, \lambda\varepsilon^*.\kappa_L(<\varepsilon> \S\varepsilon^*)))$$

We graft pattern matching on this little language thanks to the `match-all-lambda` special form. `(match-all-lambda` ϕ π`)` yields a function that takes one argument and matches it against ϕ: for each success, the body π is evaluated in the resulting environment. The pattern belongs to the pattern language exposed in section 1. An obvious improvement is to allow extended patterns and to standardize them into normalized patterns before calling $\mathcal{M}$.

$\mathcal{E}[\![\texttt{match-all-lambda}\ \phi\ \pi]\!] =$
$\lambda\rho_L\kappa_L.\kappa_L(\lambda\varepsilon^*\kappa_L'.\ \textbf{if}\ \#\varepsilon^* = 1$
$\quad\textbf{then}\ \mathcal{M}[\![\phi]\!](\varepsilon^*\downarrow_1, \rho_{init}, \alpha_{init}, \mu_{init}$
$\quad\quad, \lambda\varepsilon\rho\zeta.\mathcal{E}[\![\pi]\!](\lambda\nu.\ \textbf{if}\ \rho(\nu) = \mathit{unbound\text{-}pattern}$
$\quad\quad\quad\textbf{then}\ \rho_L(\nu)$
$\quad\quad\quad\textbf{else}\ \rho(\nu)$
$\quad\quad\quad\textbf{endif}\ , \lambda\varepsilon'.\zeta())$
$\quad\quad, \lambda().\mathit{wrong}(\text{“}\texttt{MATCH failed}\text{”}, \varepsilon^*\downarrow_1))$
$\quad\textbf{else}\ \mathit{wrong}(\text{“}\texttt{MATCH requires one argument}\text{”}, \varepsilon^*)$
$\quad\textbf{endif}\ )$

The special form `match-all-lambda` involves many effects: it suspends the normal ($\mathcal{E}$) evaluation, switches to the matching ($\mathcal{M}$) process, the continuation of which being to resume $\mathcal{E}$ on the body π of `match-all-lambda` with the current lexical environment ρ_L augmented with the pattern variable environment ρ. The continuation of the body π is to resume the matching process and find other solutions if any. When all solutions are found, `match-all-lambda` invokes *wrong* to stop the computation[7].

From `match-all-lambda`, it is simple to devise new constructs. For example and with help of Scheme's `call/cc`, `match-lambda`[8] can only computes the first match and exits with the value of its body: `(match-lambda` ϕ π`)` is equivalent to `(call/cc (lambda (`k`) (match-all-lambda` ϕ `(`k π`))))`, where k is free in ϕ and π to avoid capturing variable.

Other enhancements such as calling back Lisp from patterns will be seen later.

4 Canonical Compilation

The goal of this section is to present a compiler that takes a pattern and produces a valid program for our little functional language. $\mathcal{M}$ was in fact a pattern interpreter, the problem is to exhibit $\mathcal{C}$: the pattern compiler. We follow an idea of Eugen Neidl [Neidl 83] saying that $\mathcal{M}$ and $\mathcal{C}$ are quite similar. $\mathcal{C}$ is like $\mathcal{M}$ except where $\mathcal{M}$ matches a pattern, $\mathcal{C}$ generates the code to match this pattern. Consider the domains involved in $\mathcal{M}$ and let us change **Val** and **MAnswer** to be just **Program**, now $\mathcal{C}$ appears as:
$\mathcal{C}: \textbf{Pattern} \rightarrow \textbf{Program} \times \textbf{MEnv} \times \textbf{Seg} \times \textbf{Rep} \times \textbf{MCont} \times \textbf{Alt} \rightarrow \textbf{Program}$

[7] Since we do not want to complicate the semantics we neglect exception handling which can grasp the `"Match failed"` exception and continue the computation.

[8] Due to its widespread use, `matchlambda` could be defined directly without resorting to a construct as powerful as `call/cc`. It nevertheless simplifies the paper since there is only a single primitive: `matchall-lambda`.

where, for instance, the first occurrence of **Program** is no more the datum to match but the program excerpt which accesses the datum.

In Lisp terms the real goal is to make `match-all-lambda` a macro, so we have to find ways to convert its denotation into an equivalent program generator. The task is not so easy since off-the-scene entities like environment or continuation appear in the denotation. We therefore take two implementation decisions:

- pattern variables are implemented via lexical variables. We have in fact no other means to achieve the coercion between the matching environment and the surrounding lexical environment. The only choice concerns the coercion time, we choose here to immediately coerce to allow bindings sharing.
- failure is coded as bactrack, i.e.choice points are introduced by `OR`. To report a failure is just to return `#!FALSE`. That works since failure handling does not require the current environment. This decision was motivated by efficiency since a stack-based continuation is usually faster than heap-based closure invocation.

Thanks to the $\mathcal{C}$ compiler, (`match-all-lambda` ϕ π) will be compiled into:

$$\begin{array}{l}\underline{\texttt{(LAMBDA}}\ \underline{(}g\underline{)}\underline{\texttt{(OR}}\ \mathcal{C}[\![\phi]\!](g, \rho_{init}, \alpha_{init}, \mu_{init}, \lambda\varepsilon\rho\zeta.\underline{\texttt{(BEGIN}}\ \pi\ \underline{\texttt{\#!FALSE)}}, \lambda().\underline{\texttt{\#!FALSE}})\\ \qquad\qquad\quad \underline{\texttt{(MATCH}-\texttt{FAIL))}}\end{array}$$

The lexical variable g holds the datum: g is chosen to be free in ϕ and π. The body of the generated `lambda` is the result of the compilation of ϕ. We use the Scheme form `begin` to signify that π is evaluated then its result is discarded and `#!false` is returned instead[9]. We also use the `MATCH-FAIL` function which meaning is to invoke *wrong* when appropriate. Note that program excerpts are underlined in all these equations. $\mathcal{C}$ is then defined as:

$$\mathcal{C}[\![\texttt{*sexp}\ ]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\kappa(\varepsilon,\rho,\zeta)$$

$$\mathcal{C}[\![\texttt{*quote}\ \varepsilon]\!] = \lambda\varepsilon'\rho\alpha\mu\kappa\zeta.\underline{\texttt{(IF}}\ \underline{\texttt{(EQUAL?}}\ \varepsilon'\ \underline{\texttt{(QUOTE}}\ \varepsilon\underline{))}\ \kappa(\varepsilon',\rho,\zeta)\ \zeta()\underline{)}$$

$$\begin{array}{l}\mathcal{C}[\![\texttt{*cons}\ \phi_1\ \phi_2]\!] =\\ \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\underline{\texttt{(IF}}\ \underline{\texttt{(PAIR?}}\ \varepsilon\underline{)}\\ \qquad\qquad \mathcal{C}[\![\phi_1]\!](\underline{\texttt{(CAR}}\ \varepsilon\underline{)}, \rho, \alpha_{init}, \mu_{init}\\ \qquad\qquad\qquad , \lambda\varepsilon'\rho'\zeta'.\mathcal{C}[\![\phi_2]\!](\underline{\texttt{(CDR}}\ \varepsilon\underline{)}, \rho', \alpha, \mu, \kappa, \zeta'), \zeta)\ \zeta()\underline{)}\end{array}$$

$$\mathcal{C}[\![\texttt{*and}\ \phi_1\ \phi_2]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{C}[\![\phi_1]\!](\varepsilon,\rho,\alpha,\mu,\lambda\varepsilon'\rho'\zeta'.\mathcal{C}[\![\phi_2]\!](\varepsilon,\rho',\alpha,\mu,\kappa,\zeta'),\zeta)$$

The previous definitions are directly equivalent to their $\mathcal{M}$ counterparts. Since we code failure as backtrack, the failure continuation of the first pattern in `*or` returns false, the backtrack is then performed by the surrounding `OR`. The same trick is applied on `*not` definition: success or failure continuations just return a boolean.

$$\begin{array}{l}\mathcal{C}[\![\texttt{*or}\ \phi_1\ \phi_2]\!] =\\ \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\underline{\texttt{(OR}}\ \mathcal{C}[\![\phi_1]\!](\varepsilon,\rho,\alpha,\mu,\kappa,\lambda().\underline{\texttt{\#!FALSE}})\ \mathcal{C}[\![\phi_2]\!](\varepsilon,\rho,\alpha,\mu,\kappa,\zeta)\underline{)}\end{array}$$

$$\begin{array}{l}\mathcal{C}[\![\texttt{*not}\ \phi]\!] =\\ \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\underline{\texttt{(IF}}\ \mathcal{C}[\![\phi]\!](\varepsilon,\rho,\alpha,\mu,\lambda\varepsilon'\rho'\zeta'.\underline{\texttt{\#!TRUE}},\lambda().\underline{\texttt{\#!FALSE}})\ \zeta()\ \kappa(\varepsilon,\rho,\zeta)\underline{)}\end{array}$$

According to our first implementation decision, the binding for the matching variable is immediately performed, so does `*setq` and also `*ssetq-append`. The others (`*eval`, `*eval-append`, `check` and `cut`) are similar to their $\mathcal{M}$ counterpart.

$$\begin{array}{l}\mathcal{C}[\![\texttt{*setq}\ \nu\ \phi]\!] =\\ \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{C}[\![\phi]\!](\varepsilon,\rho,\alpha,\mu,\lambda\varepsilon'\rho'\zeta'.\ \begin{array}[t]{l}\textbf{if}\ \rho(\nu) = \textit{unbound-pattern}\\ \textbf{then}\ \underline{\texttt{(LET}}\ \underline{((}\nu\ \varepsilon\underline{))}\ \kappa(\varepsilon,\rho'[\nu\to\nu],\zeta')\underline{)}\\ \textbf{else}\ \textit{wrong}(\text{“}\texttt{Cannot rebind pattern}\text{”},\nu)\\ \textbf{endif}\ ,\zeta)\end{array}\end{array}$$

---

[9] `begin` (or `progn` in Lisp) expresses the sequentiality.

$\mathcal{C}[\![\texttt{*eval}\ \nu]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.$ **if** $\rho(\nu) = \textit{unbound-pattern}$
then $\textit{wrong}($"`Unbound pattern`"$, \nu)$
else $\underline{\texttt{(IF}}\ \underline{\texttt{(EQUAL?}}\ \rho(\nu)\ \varepsilon\underline{)}\ \kappa(\varepsilon,\rho,\zeta)\ \zeta()\underline{)}$
endif

$\mathcal{C}[\![\texttt{*ssetq-append}\ \nu\ \phi_1\ \phi_2]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{C}[\![\phi_1]\!](\varepsilon, \rho, \alpha_{init}[\nu \rightarrow \lambda\varepsilon'\rho'\zeta'.$ **if** $\rho'(\nu) = \textit{unbound-pattern}$
then $\underline{\texttt{(LET}}\ \underline{((}\nu\ \underline{\texttt{(CUT}}\ \varepsilon\ \varepsilon'\underline{)))}$
$\mathcal{C}[\![\phi_2]\!](\varepsilon', \rho'[\nu \rightarrow \nu], \alpha, \mu, \kappa, \zeta')\underline{)}$
else $\textit{wrong}($"`cannot rebind`"$, \nu)$ **endif** $]$
$, \mu_{init}, \lambda\varepsilon'\rho'\zeta'.\textit{wrong}($"`Ssetq not ended`"$, \phi_1), \zeta)$

$\mathcal{C}[\![\texttt{*eval-append}\ \nu\ \phi]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.$ **if** $\rho(\nu) = \textit{unbound-pattern}$
then $\textit{wrong}($"`Unbound segment`"$, \nu)$
else $\underline{\texttt{(CHECK}}\ \varepsilon\ \rho(\nu)\ \underline{\texttt{(LAMBDA}}\ \underline{(}g\underline{)}\ \mathcal{C}[\![\phi]\!](g, \rho, \alpha, \mu, \kappa, \zeta)\underline{)}\ \underline{\texttt{(LAMBDA}}\ \underline{()}\ \zeta()\underline{))}$
endif

The library functions `cut` and `check` are essentially the same as the semantical functions *cut* and *check* shifted to the level of our functional language:

```
(DEFINE (CHECK E EE FN Z)
  (IF (AND (PAIR? E)
           (PAIR? EE)
           (EQUAL? (CAR E) (CAR EE)) )
      (CHECK (CDR E) (CDR EE) FN Z)
      (IF (NULL? EE) (FN E) (Z)) ) )
```

```
(DEFINE (CUT E EE)
  (IF (EQ? E EE) '()
    (CONS (CAR E)
          (CUT (CDR E) EE) ) ) )
```

$\mathcal{C}[\![\texttt{*end-ssetq}\ \nu]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\alpha(\nu)(\varepsilon, \rho, \zeta)$

Repetitions introduced by `*times` are the most difficult to transliterate. The *try* auxiliary function is turned into a monadic function which takes the datum. Due to how we code matching environments we must take care of the lexical environment where `*end-times` is called. If it is the same then a recursive call to *try* is perfect otherwise we have to unfold the `*times` pattern. This eventually finishes since there is a finite number of pattern variables. This case correspond to kind of teratological patterns (forbidden in [Heckmann 88]) such as:

```
(*times A (*cons (*or (*setq X (*sexp)) (*setq Y (*sexp)))
                 (*end-times A) )
          (*quote ()) )
```

This pattern can only accept list of zero, one or two terms since it is an error to rebind a variable.

$\mathcal{C}[\![\texttt{*times}\ \nu\ \phi_1\ \phi_2]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.\underline{(\texttt{(REC}}\ \textit{try}\ \underline{\texttt{(LAMBDA}}\ \underline{(}g\underline{)}$
$\underline{\texttt{(OR}}\ \mathcal{C}[\![\phi_2]\!](g, \rho, \alpha, \mu, \kappa, \lambda().\underline{\texttt{\#!FALSE}})$
$\mathcal{C}[\![\phi_1]\!](g, \rho, \alpha_{init}$
$, \mu_{init}[\nu \rightarrow \lambda\varepsilon'\rho'\zeta'.$ **if** $\rho' = \rho$
then $\underline{(}\textit{try}\ \varepsilon'\underline{)}$
else $\mathcal{C}[\![\texttt{*times}\ \nu\ \phi_1\ \phi_2]\!]$
$(\varepsilon', \rho', \alpha, \mu, \kappa, \zeta')$
endif $]$
$, \lambda\varepsilon'\rho'\zeta'.\textit{wrong}($"`Times not ended`"$, \phi_1), \zeta)\underline{)))}\ \varepsilon\underline{)}$

$\mathcal{C}[\![\texttt{*end-times}\ \nu]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mu(\nu)(\varepsilon, \rho, \zeta)$

This compiler is very naive. Its only merit is to be very close from the semantics of patterns as provided by $\mathcal{E}$ and thus improves the confidence one can put in it. Some examples appear in the following sections.

Since we provided two different semantics for our pattern language: the interpreter $\mathcal{M}$ and the compiler $\mathcal{C}$, it is natural to expect these two semantics to be congruent. The proposition we want to prove, with g free in π, is:

$$\mathcal{E}[\![\texttt{(match-all-lambda}\ \phi\ \ \pi\texttt{)}]\!] \equiv$$
$$\mathcal{E}[\![\underline{\texttt{(LAMBDA}}\ \underline{(}g\underline{)}\underline{\texttt{(OR}}\ \mathcal{C}[\![\phi]\!](g, \rho_{init}, \alpha_{init}, \mu_{init}, \lambda\varepsilon\rho\zeta.\underline{\texttt{(BEGIN}}\ \pi\ \underline{\texttt{\#!FALSE)}}, \lambda().\underline{\texttt{\#!FALSE)}}$$
$$\underline{\texttt{(MATCH-FAIL))}}]\!]$$

The proof seems to be very long and to involve multiple steps. It uses some extra notions not given in this paper such as the $\mathcal{E}$-semantics of `or`, `and`, `let` as well as `match-fail`, `pair?`, `equal?` not omitting `#!true` and `#!false`. As a future work we will try to mechanically obtain the proof, probably with help of partial evaluation.

5 Compilation Variants

Many variants or improvements can be brought to the previous compiler.

$\gg$ If we want to add pattern matching to a language such as Lisp, we ought to offer some way to call back Lisp from within patterns. We thus add two new patterns `*check` (resp. `*success`) which component is a predicate (resp. a form). `*check` only accepts a datum that satisfies its predicate; `*success` accepts the current datum if the form does not yield `#!false`. This very form is evaluated in the current lexical context where some pattern variables might be bound. Their definition is straightforward:

$$\mathcal{C}[\![\texttt{*check}\ \pi]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\underline{\texttt{(IF}}\ \underline{(}\pi\ \varepsilon\underline{)}\ \kappa(\varepsilon,\rho,\zeta)\ \zeta()\underline{)}$$

$$\mathcal{C}[\![\texttt{*success}\ \pi]\!] = \lambda\varepsilon\rho\alpha\mu\kappa\zeta.\underline{\texttt{(IF}}\ \pi\ \kappa(\varepsilon,\rho,\zeta)\ \zeta()\underline{)}$$

The pattern `(*success` π`)` is perfect for side-effects and allows to introduce new abstractions over `match-all-lambda` such as `(match-case` π `(`ϕ_1 π_1`) ... (`ϕ_n π_n`))` which mimics the ML `case`: π is evaluated then matched against ϕ_1, if success `match-case` returns what returns the evaluation of π_1, if failure then the value of π is matched against ϕ_2 and so on ...`match-case` is expanded into:

```
(lambda (ε)
  (call/cc (lambda (k)
             ((match-lambda (*or (*and φ1 (*success (k π1)))
                                 (*or ...
                                      (*and φn (*success (k πn)))
                                                ... ) )
                       (match-fail) )
               ε ) )) )
```

Also note that `(match-all-lambda` ϕ π`)` is, thanks to `*success`, equivalent to `(match-all-lambda (*and` ϕ `(*success (begin` π `#!false))) (matchfail))`. This form simplifies the management of the body of `match-all-lambda`.

$\gg$ The failure continuation is always the same everywhere except within `*eval-append`. We can eliminate this remaining case since $\lambda().\zeta()$ is η-equivalent to ζ and write:

$\mathcal{C}[\![\texttt{*eval-append}\ \nu\ \phi]\!] =$
$\lambda\varepsilon\rho\alpha\mu\kappa\zeta.$ **if** $\rho(\nu) = \textit{unbound-pattern}$
then $\textit{wrong}(\text{“}\texttt{Unbound segment}\text{”}, \nu)$
else $\underline{\texttt{(NCHECK}}\ \varepsilon\ \rho(\nu)\ \underline{\texttt{(LAMBDA}}\ \underline{(}g\underline{)}\ \mathcal{C}[\![\phi]\!](g,\rho,\alpha,\mu,\kappa,\zeta)\underline{)}\ \underline{)}$
endif

This must be associated to the new version of `CHECK` called `NCHECK`:

```
(DEFINE (NCHECK E EE FN)
  (IF (AND (PAIR? E)
           (PAIR? EE)
           (EQUAL? (CAR E) (CAR EE)) )
      (NCHECK (CDR E) (CDR EE) FN)
      (IF (NULL? EE) (FN E) #!FALSE) ) )
```

It is now possible to entirely eliminate ζ from all the definitions. The meaning of that is that failure is now encoded in the continuation κ_L provided by the underlying language i.e.the stack.

$\gg$ All techniques available for the compilation of functional language such as lambda-lifting [Peyton Jones 86] or lambda-hoisting [Takeichi 88] can be applied. To introduce the assignment also allows to reuse some existing bindings but closures created inside `*success` or `*check` patterns might then share bindings.

$\gg$ The generated code performs a lot of accesses (long chains of `car` and `cdr`), many comparisons (with `equal`) and some allocations with the `cut` form to build segments. One can expect the compiler of the underlying language to minimize the cost of accesses. Comparisons seem unavoidable but we can defer the costful computation of (`cut` $\varepsilon\ \varepsilon'$) until really needed and write something like:

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{*ssetq-append}\ \nu\ \phi_1\ \phi_2]\!] = \\
&\lambda\varepsilon\rho\alpha\mu\kappa\zeta.\mathcal{C}[\![\phi_1]\!](\varepsilon,\rho,\alpha_{init}[\nu \rightarrow \lambda\varepsilon'\rho'\zeta'.\ \textbf{if}\ \rho'(\nu) = \textit{unbound-pattern} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (\texttt{LET}\ ((\nu\ (\texttt{DELAY}\ (\texttt{CUT}\ \varepsilon\ \varepsilon')))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{C}[\![\phi_2]\!](\varepsilon',\rho'[\nu \rightarrow (\texttt{FORCE}\ \nu)] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ,\alpha,\mu,\kappa,\zeta')) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \textit{wrong}(\text{“}\texttt{cannot rebind}\text{”},\nu) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif}\,] \\
&\qquad\qquad ,\mu_{init},\lambda\varepsilon'\rho'\zeta'.\textit{wrong}(\text{“}\texttt{Ssetq not ended}\text{”},\phi_1),\zeta)
\end{aligned}
$$

We neglect the problem we just introduced since variable ν must be explicitely `force`-d in the body of `*success` patterns or similarly in the body of `match-all-lambda`. This may be solved by an appropriate code-walker [Dybvig, Friedman & Haynes 88] or alternatively, pattern variables can be automatically `force`-d before `*success` or `*check` patterns.

One other improvement is to avoid to compute (`cut` $\varepsilon\ \varepsilon'$) for segment comparison as currently done in `*eval-append`. We may directly use the head and tail of the segment and rewrite (`NCHECK` ε (`CUT` $\varepsilon'\ \varepsilon''$) ϕ) into (`SCHECK` $\varepsilon\ \varepsilon'\ \varepsilon''\ \phi$) with:

```
(DEFINE (SCHECK E HEAD TAIL FN)
  (IF (EQ? HEAD TAIL) (FN E)
      (IF (PAIR? E)
          (IF (EQUAL? (CAR E) (CAR HEAD))
              (SCHECK (CDR E) (CDR HEAD) TAIL FN)
              #!FALSE )
          #!FALSE ) ) )
```

With these modifications a pattern such as `(match-lambda (??x ??x) x)` only calls `cut` once with a correct solution.

$\gg$ We confuse in this paper segment variables with term variables i.e.variables bound by `*setq` or `*ssetq-append`. This does not seem to be harmful in a world with homogeneous lists, it hurts in Lisp since a segment variable is a proper list (i.e.a dotted pair which final `cdr` is the empty list) unlike a term variable which can hold any kind of value and especially non proper lists. In other words (`??`x) is not equal to `?`x. The semantics can be refined to exclude mixing these kinds of variables.

The `?-` pattern differs from `(??-)` on both semantical and efficiency aspects. The latter checks that the ultimate tail of the datum is the empty list. To avoid this we introduce, for Lisp or Scheme, the extended patterns `???-` and `???`x which may be better compiled if in tail position. `(foo ???x)` is standardized into `(*cons (*quote foo) (*setq x (*sexp)))`. It may thus be erroneous to refer to such a variable as a list since it can hold an improper list for value.

≫ Another problem is wether comparisons are performed with `equal` or `eq`. These two predicates test, in Lisp, equality or physical identity. We here chose `equal` but more often pattern matchers let the user choose between the two. One may add new patterns to offer comparisons with physical identity.

≫ We neglect the problem of the readability of the generated code. This can be improved by well known code-walking techniques. Incidentally pattern matching is, associated to its near reciprocal: the backquote notation, a very convenient tool for these rewriting transformations. For instance to translate `(if` π π' `#!false)` into `(and` π π'`)` is expressed by:

```
(match-lambda (if ?x ?y #!false) `(and ,x ,y))
```

The readability is a major concern for the implementer to debug the compiler (or for the user to gain confidence in it). It is not very useful for the compiler since most of the times `and`, `or`, composition of `car` and `cdr` forms are reexpanded before compilation.

≫ Observe that the success continuation may be multiply invoked in different contexts. For instance:

```
(match-lambda (*cons (*or (*quote foo)   ;((*or foo bar) ?x)
                          (*quote bar) )
                     (*setq x (*sexp)) )
      π )
```

is compiled into

```
(lambda (e) (if (pair? e)
                (or (if (equal? (car e) 'foo)
                        (let ((x (cdr e))) π)   ;repeated excerpt
                        #!false )
                    (if (equal? (car e) 'bar)
                        (let ((x (cdr e))) π)   ;repeated excerpt
                        #!false ) )
                #!false ) )
```

The repeated excerpt can be much more voluminous than this example so common subexpressions elimination is highly necessary. This improvement must not be done totally blindly since common subexpressions may appear in different lexical context. Consider for example: `(match-lambda (*or ?x ?y)` π`)` which is compiled into

```
(lambda (e) (or (let ((x e)) π)
                (let ((y e)) π) ) )
```

≫ This latter example also sets the problem of wether it bears some meaning to have pattern disjunctions on different sets of pattern variables. At first glance we might consider it as strange but the `match-case` expansion (see above) as the case analyses in ML take it for granted. To accept it means that one must also accept that some capture might appear. For example

```
(let ((x "weird") (y "queer"))
     ((match-all-lambda (*or ?x ?y) (print (list x y)))
      "thing" ) )
;;yields
(thing queer)
(weird thing)
```

A similar problem is `(?x ...)` that accepts lists which elements are all the same. It accepts `()` in which case `x` is not bound. It also accepts `(foo foo)` which binds `x` to `foo`. The `(?x ...)` pattern may be rewritten into `(*or () (*cons ?x (?x ...)))` which brings us back into the previous problem since the `*or` branches are not equilibrated.

In our pattern match compiler prototype we decide to warn the user but only during the standardization[10] not during the compilation.

[10] The standardization is the first phase of the compiler where the original pattern is converted from an extended and convenient syntax to the reduced pattern language we defined in section 1.

6 Code Generation Examples

Let us just give one example of code generation on the rather theoretical pattern `(??x ??x)`. We incorporate all the improvements we discussed plus others like the use of assignment within `delay`-ed expressions. The code can still be improved but we just want to demonstrate the behavior of an existing refined version deriving from the simple compiler described above.

```
(match-lambda (??x ??x) x) ;;is expanded into
(lambda (G4)
  (call/cc
   (lambda (G5)
     (letrec
        ((G7 (lambda (G8)
               (or (letrec ((G9 (delay (set! x (cut G4 G8))))
                            (x 'wait) )
                     (scheck G8 G4 G8
                             (lambda (G0)
                               (and (equal? G0 '())
                                    (and (begin (force G9)
                                                (G5 (begin x)) )
                                         #!false ) ) ) ) ) )
                   (and (pair? G8) (G7 (cdr G8))) ) )))
       (G7 G4) ) ) ) ) )
```

We also wrote a bigger compiler for Common Lisp. Two phases (standardization, translation) composes the compiler. The standardization deals with the difficult syntactic issue: how to conveniently write patterns. We introduce a notion of "*macro-pattern*" which meaning is expressed by a rewrite rule. For instance:

```
(defmacro-pattern (*fail) `(*not ?-))
```

defines a term-pattern, i.e.a pattern which can only match an unique datum, which always fails. The new term-pattern named `(*fail)` is defined as an abbreviation of `(*not (*sexp))`.

Another example is:

```
(defmacro-pattern ((*maybe something) . patterns)
  `(*or (,something . ,patterns) ,patterns) )
```

This latter pattern is a segment pattern, i.e.a pattern that can accept a sequence of data, which takes an argument *something* and a right context *patterns*. Used inside a list of patterns, `*maybe` accepts any datum matched by *patterns* that may be prefixed by a term accepted by *something*.

The standardization translates the original pattern until it belongs to the reduced pattern language. Then this pattern is compiled as explained above.

7 Related Works and Conclusions

Much work has been done on pattern matching. The most related one is probably [Heckmann 88] who proposes a pattern language which power is quite similar to ours. He gives the semantics of its pattern language with a non deterministic style. He introduced a pattern (noted *pattern1*$\uparrow$*pattern2*) which allows to create tree-fragments (trees with a single hole within them). We do not offer this powerful pattern but provide a richer segment assignment mechanism as well as full and complete compilation of our patterns.

Plasma heavily uses pattern matching with segments but there has been not so much work on its compilation [Arcangeli & Pomian 90]. We improve on this since segment patterns of Plasma are restricted to `??`x or `??-`, where we allow, for instance, `((??- ?x ??-) ...)` which accepts all list composed of sublists having at least one common term.

Other works mainly focus on the compilation of patterns excluding segments. Intense efforts have been spent to perfectly compile alternative patterns sharing common prefixes [Wadler 86]. These transformations may be implemented in our standardization phase: `(*or (?x` ϕ_1`) (?x` ϕ_2`))` may be rewritten as `(?x (*or` ϕ_1 ϕ_2`))`.

Two main uses of pattern matching can be recognized. The first use is case analysis and scoped destructuration as offered in ML-like languages. The second use, taking advantage of lists and segments, is helpful for macros and code-walkers in Lisp-like languages. Macros usually introduce some syntax that can easily be parsed by appropriate patterns. Segments patterns like $??x$ often correspond to the `&rest` or `&body` binding keyword of COMMON LISP.

On the side of segment variables, one may object that they are more efficient algorithm to handle the `(??x ??x)` pattern. We almost never saw a reference to a segment variable: patterns like `(??x ??x)`, where the user repeats a sequence of terms, seem to be purely theoretical. We may thus be suboptimal in their compilation.

Pattern matching is to a large extent language-independent. We introduced a simple but powerful intermediate pattern language which allows to express boolean conjunctions, disjunctions and negations as well as segment and repetition handling. We defined a very naive and functional compiler translating these patterns into functional code that can be easily retargetted to other functional languages. We discussed several improvements of the generated code and commented several examples of use.

Acknowledgments

We wish to thank Pierre Weis who converts this compiler from Scheme to CAML and Olivier Danvy who points out that partial evaluation can be valueful to prove the equivalence proposition. We also wish to thank the numerous referees for their helpful work.

Bibliography

[Abelson & Sussman 85] Harold Abelson, Gerald Sussman, with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge MA, 1985.

[Arcangeli & Pomian 90] Jean-Paul Arcangeli, Christian Pomian, *Principles of Plasma pattern and alternative structure compilation*, Theoretical Computer Science, Vol 71, 1970, pp 177-191.

[Augustsson 85] Lennart Augustsson, *Compiling Pattern Matching*, Conference on Functional Programming and Machine Architecture, Nancy 1985, Lecture Note in Computer Science 201, Springer Verlag 1985.

[Burstall 69] R. M. Burstall, *Proving Properties of Programs by structural Induction*, The Computer Journal, Vol 12, N° 1.

[Dybvig, Friedman & Haynes 88] R. Kent Dybvig, Daniel P. Friedman, Christopher T. Haynes, *Expansion-Passing-Style: A General Macro Mechanism*, Lisp and Symbolic Computation, Vol 1, n 1, June 1988, pp 53-76.

[Heckmann 88] Reinhold Heckmann, *A Functional Language for the Specification of Complex Tree Transformations*, ESOP 88, Lecture Notes on Computer Science, Springer 1988.

[Hewitt & Smith 75] C. Hewitt, B. Smith, *Towards a programming Apprentice*, IEEE Transactions on Software Engineering, SE-1, N° 1, March 1975, pp 26-45.

[Hudak & Wadler 90] Paul Hudak & Philip Wadler (eds), *Report on the Programming Language Haskell*, YALEU/DCS/RR-777, 1 April 1990.

[Kessler 88] Robert R. Kessler, *Lisp, Objects, and Symbolic Programming*, Scott, Foreman/Little, Brown College Division, Glenview, Illinois, 1988.

[Laville 88] Alain Laville, *Évaluation paresseuse des filtrages avec priorité, Application au langage ML*, Thèse de Doctorat, Université Paris VII, Février 1988.

[Neidl 83] Eugen Neidl, *Étude des relations avec l'interprète dans la compilation de Lisp*, Thèse de troisième cycle, Université Paris VI, 1984.

[Peyton Jones 86] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, May 1986.

[Queinnec 84] Christian Queinnec, *Lisp*, Macmillan, 1984.

[Rees & Clinger 86] Jonathan A. Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

[Schmidt 86] David A. Schmidt, *Denotational Semantics, A Methodology for Language Development*, Allyn and Bacon, Inc., Newton, Mass., 1986.

[Stoy 77] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

[Takeichi 88] M. Takeichi, *Lambda-Hoisting: a Transformation technique for fully lazy evaluation of functional program*, New Generation Computing, Vol 5, 1988, pp 377-391.

[Teitelman 78] Warren Teitelman, *InterLisp Reference Manual*, 1978.

[Turner 76] D. A. Turner, *The SASL Language Manual*, Technical Report CS/75/1, Department of Computational Science, University of Saint Andrews.

[Wadler 86] Philip Wadler, *Efficient Compilation of Pattern-Matching, in* PeytonJones86.

[Weis 89] Pierre Weis, *The CAML Reference Manual*, INRIA 1989.

[Winston 88] Patrick H. Winston, Berthold K. Horn, *Lisp*, 3rd Edition, Addison Wesley, 1988.