Typing deep pattern-matching in presence of polymorphic variants

Jacques Garrigue†

Polymorphic variants are a well-known feature of the Objective Caml programming language, and they have turned popular since their introduction. They allow structural equality of algebraic type definitions, and code reuse through their polymorphism.

Their typing and compilation have been studied in the past, and there are already detailed published works for both[2),4)]. By their very nature, polymorphic variants depend on pattern-matching to analyze their contents. However, only typing for shallow pattern-matching was studied in the past. In that case, checking exhaustiveness is trivial, and the natural typing rule guarantees it.

Deep pattern-matching is more complex, as other constructors may appear nested in the same pattern-matching. Exhaustiveness check is available, but only after finishing type checking, while we would like to use it to define the typing of polymorphic variant patterns. We explain the tradeoffs, and define a type checking algorithm for pattern-matching containing polymorphic variants which is symmetric.

1. Introduction

The name *polymorphic variant* stands for a particular kind of sum type whose constructors are automatically inferred by the type checker. This automatic inference is not only useful because it avoids writing a type definition — actually one could argue that writing a type definition is a good idea anyway—, but also because it allows to use the same constructor inside different types, and allows structural equality and subtyping between variant types[2),4)].

The strict binding of a constructor to a specific type enforced by ML algebraic datatypes is sometimes burdensome. Think for instance of the multiple intermediate data structures you have in a staged compiler. They are often very similar in structure, but because of some small differences, all their constructors must have different names (so we end up with an `Avariable` and a `Bvariable`, an `Afunction` and a `Bfunction`, ...), and all code working on these data structures must be rewritten for each type.

With polymorphic variant types, you get more freedom: you can reuse the same constructor name without fear of conflict, and you can even reuse pieces of code working on common sets of constructors. A complete example of such code reuse has been described[3)].

While the main interest of polymorphic variants resides in their typing, a subtle part of the problem had not been seriously studied until now: the interaction between pattern-matching and polymorphic variant typing. To be more specific, all accounts of polymorphic variant only consider flat pattern-matching, in which the choice of the branch depends only on the constructor, and each constructor occurs only once. In that case, the list of handled constructors (and the upper bound of the corresponding polymorphic variant type) comes from a direct reading of the pattern-matching. However, if we think of deep pattern-matching, where polymorphic variant constructors may occur deeper in the structure, then which constructors are handled in all cases is no longer trivial, and we must be very careful about how we type pattern-matching, lest we loose symmetry (for instance changing the order of the cases in a pattern-matching could lead to a different type), or even principality of the whole type system (typing of other parts of a program could influence the typing of patterns in a non-monotonic way).

As a matter-of-fact, the problem of typing pattern-matching has rarely been studied. In most cases, it is just subsumed by the typing of isomorphic expressions. Outside of polymorphic variants, the only specific constructs that come to mind are Standard ML record ellipsis[7)], or Objective Caml's variables in or-patterns, and they are not really difficult. Our concern with deep matching is related to exhaustiveness check[6)], yet it differs from this previous work (unfortunately only available in French)

† 京都大学数理解析研究所
Kyoto University Research Institute for Mathematical Sciences

in that polymorphic variants are a new feature, and that we need exhaustiveness information to refine our types, while in general exhaustiveness depends on the type —hence we must avoid a semantic loop. On the other hand, while the main problem of pattern-matching, namely compilation, has been studied for a long time already[1),9)], it is not directly related to our endeavor: once properly typed, polymorphic variants can easily be combined in a compilation algorithm[2)].

After giving some basic notions of polymorphic variants and their typing, we will see the problems caused by deep pattern-matching, and how they can be solved. We will then give a formal account of our solution.

2. Polymorphic variants basics

Polymorphic variants are a standard feature of Objective Caml[5)] since version 3. To make our account more intuitive we will use examples typed by the Objective Caml toplevel. We will not enter here into the details of the type system, which can be found in other papers[4)].

Polymorphic variants are particular in that a different type is constructed for each value introduced.

```
let a = `Apple
val a : [> `Apple]
let b = `Orange("Spain")
val b : [> `Orange of string]
```

The backquote "`" indicates a polymorphic variant constructor. When used without argument, the type indicates just that this constructor is present. When there is an argument to the constructor, its type appears in the polymorphic variant type.

```
let l = [a; b]
val l :
  [> `Apple | `Orange of string] list
```

If you define a list containing apples and oranges, then it becomes a list of apples and oranges. There is actually a type variable hidden behind the ">" in the above types, and copies of the original types can be unified to obtain a larger type.

The symmetric of constructor introduction is destruction through pattern-matching.

```
let show = function
    `Apple -> "apple"
  | `Orange s -> "orange/" ^ s
val show :
  [< `Apple | `Orange of string] -> string
let l' = List.map show l
val l' : string list =
  ["apple"; "orange/Spain"]
```

As you can see, the type inferred for pattern-matching starts with a "<" rather than a ">". This denotes a list of accepted variant constructors, with the types of their arguments. Again the type is polymorphic, and can be applied to several kinds of variant constructors. Acceptor types can be unified together, resulting in a type accepting less constructors. Presence types and acceptor types can be unified, as long as all present constructors are also accepted.

```
let id = function
    `Apple -> 1
  | `Orange _ -> 2
  | `Pear -> 3
val id :
  [< `Apple | `Orange of 'a | `Pear ] -> int
let f x = (show x, id x)
val f :
  [< `Apple | `Orange of string ] ->
  string * int
```

While `id` accepts more constructors than `show`, applying both to the same variable results in a smaller type.

```
let g x = if (id x = 3) then `Apple else x
val g :
  ([< `Apple | `Orange of 'b | `Pear
    > `Apple ] as 'a) -> 'a
```

Since it appears in both input and output, the type of `x` contains both acceptor and presence information. The `as` denotes sharing.

Finally, the ability of acceptor types to loose members makes necessary the introduction of *conjunctive types* in variant arguments.

```
let id2 = function
    `Apple -> 1
  | `Orange n -> n
val id2 :
  [< `Apple | `Orange of int ] -> int
let h x = (show x, id2 x)
val h :
  [< `Apple | `Orange of int & string ] ->
  string * int
```

The type `int & string` means that, in order to be accepted by `h`, an orange should have an argument of type *both* `int` and `string`. This is of course impossible, which limits possible inputs to apples. From an inference point of view, all members of a conjunctive type have to be unified when its constructor becomes present. Conjunctive types are only useful in intermediate steps of type inference, to make sure that unification is associative, and that principality

is kept.

3. Closed and open matching

All the functions we have considered up to now have used closed pattern-matching. That is, an exhaustive list of accepted constructors was provided. However, it is not rare for pattern-matchings to use "wild cards" to catch all remaining cases.

```
let show3 = function
    `Apple -> "apple"
  | `Peer  -> "pear"
  | _      -> "unknown"
```

Coherently with the usual behavior of pattern-matching, the meaning of "_" can be seen as "all other variant constructors". We expect such a function to accept any variant value, except (`Apple e) and (`Peer e) (*i.e.* `Apple or `Peer used with an argument.) However this still leaves us with two possibilities. The most intuitive one may be:

```
val show3 : [< `Apple | `Peer | .. ]
```

This stands for "show3 accepts `Apple and `Peer, and any other constructor". Yet, experience proved that this was a bad idea: this type is too "weak". Some clearly erroneous programs are accepted, with their expected type.

```
let j x = (show3 x, id x)
val j :
  [< `Apple | `Orange of 'a | `Pear ] ->
  string * int
let it = show3 `Pear
val it : string * int = ("unknown", 3)
```

If we assume that `Peer was a typo for `Pear, then this answer is incorrect.

An alternative, and actually simpler possibility is to force all constructors to be present.

```
val show3 : [> `Apple | `Peer]
```

This avoids the above problem, as constructors in open variants can no longer "vanish". This is simpler, because we can limit types to acceptor types (with a finite bound), presence types, and combined types. This is also the type that we would obtain for an implementation of show3 using an association list. For these reasons, the latter typing was chosen in Objective Caml.

4. Deep matching

If we limit ourselves to flat matching, examples in the previous sections describe almost all possibilities, and it seems that no problems would arise.

However, with deep pattern-matching, the typing of pattern-matching needs to be further refined.

4.1 Conjunctive types

In the experimental version 2.99 of Objective Caml, pattern-matching was typed just as normal terms.

```
let f = function
    (true, `A x) -> `A x
  | (false, `A x) -> `B x
  | (_, `B x) -> `B x
val f :
  bool * [< `A of 'a & 'b | `B of 'b]
  -> [> `A of 'a | `B of 'b]
```

The resulting type was correct, but not intuitive: it means that the `A case will only be usable if 'a and 'b can be bound to the same type. While conjunctive types are useful in general, here they contradict the assumption that, in a pattern-matching, all cases should be usable. This only delays errors. The approach in more recent versions of Objective Caml is to first type the patterns independently of environing expressions (in order to keep principality), and disallow conjunctive types during this phase. Patterns are then unified with the type of the matched expression, this time allowing conjunctive types. As a result, we obtain the following typing.

```
val f :
  bool * [< `A of 'a | `B of 'a]
  -> [> `A of 'a | `B of 'a]
```

All conjunctive type variables have been collapsed to one, and we obtain a more intuitive typing.

4.2 Open or not open

With flat matching, the distinction between closed and open pattern-matchings is easy: this is a purely syntactical problem. Deep pattern-matching introduces grey cases.

```
let f = function
    true, `A -> 1
  | true, `B -> 2
  | false, _ -> 3
```

From a purely syntactic point of view, this pattern-matching is open: it contains a wild card, which actually matches variants. However, if we look at the first column, we see that only `A and `B will be accepted when the first component is true.

The real question here appears to be: what does "_" mean in the context of polymorphic variants? We can base ourselves on two different analogies:

- The string type: the set of all possible

variant constructors is infinite, and "_" indicates all of them, so a pattern including a wild card must be open.
- Usual sum types: for any practical application, the set of intended constructors is finite, and "_" indicates all other constructors in a finite set, which should be inferred when possible.

Interestingly, most people seem to find the first option to be more intuitive. Its main advantage is that the meaning of "_" is independent from the context. Yet, this solution is weak: the type inferred for `f` would be

```
Warning: this pattern-matching is not
exhaustive. Here is an example of a value
that is not matched:
 (true, `AnyExtraTag)
val f : bool * [> `A | `B] -> int
```

Moreover, it becomes impractical when polymorphic variants are mixed with algebraic datatypes.

```
let g = function
    `A :: _ -> 1
  | `B :: _ -> 2
  | []      -> 3
```

Here "_" means "any list", but the first interpretation makes it mean "a list containing any variant". Again, this would result in a non-exhaustive pattern-matching.

Considering these problems, we choose the second option. It should make us able to infer the following types, preserving exhaustiveness.

```
val f : bool * [< `A | `B] -> int
val g : [< `A | `B] list -> int
```

But we have not yet explained how we obtain these types. Indeed, the second option suggests "inferring" the set of constructors associated to a type, but does not define how to do it. Clearly, exhaustiveness has something to do with it: by closing some types we were able to make some pattern-matchings exhaustive.

A possible approach would then be: enforce exhaustiveness by restricting the type of polymorphic variants when they are breaking exhaustiveness. This is actually our ideal goal. However this cannot be used as definition: there are too many ways to restrict types.

```
let h = function
    `A, _  -> 1
  | `B, _  -> 2
  | _, `A  -> 3
  | _, `B  -> 4
```

In this case, either of these two types

```
val h : [< `A | `B] * [> `A | `B] -> int
val h : [> `A | `B] * [< `A | `B] -> int
```

is enough to ensure exhaustiveness. Deciding to use one of the two would loose symmetry: the typing would become dependent on the order of the patterns in the pair, and the order of the rules in the matching. The first type would also make the last two cases unusable, which seems to go against the wishes of the programmer.

If we cannot choose between these two types, then we should either close both sides (making the last two cases unusable), or keep both sides open (making the pattern-matching non-exhaustive). We could discuss at length on which of the two is worse, insisting that non-exhaustiveness denotes fundamentally a bug of the program. However, what matters here is not whether the style of this program is good or bad (there are warnings for that), but what was the intent of the program. From that point of view, making a case unusable clearly departs from the intent of the program, and we should avoid it as much as possible. As a result, in this particular case, we shall keep both sides open.

```
Warning: this pattern-matching is not
exhaustive. Here is an example of a value
that is not matched:
 (`AnyExtraTag, `AnyExtraTag)
val h : [> `A | `B] * [> `A | `B] -> int
```

After considering all these examples, now comes the time to define a clear set of rules defining what type should be given to polymorphic variants in a pattern. This will be done formally in the next section, but we give here the basic steps.

(1) Type each pattern in the pattern-matching, and unify the obtained types. In this step, all variants have open (presence) types.

(2) Build the exhaustiveness matrix corresponding to the inferred type.

(3) For each column of the matrix corresponding to a variant, check whether it allows extra constructors (*i.e.* the type is open). If it does, then, assuming that all other variant columns have closed types (limited to the constructors inferred in step 1), check whether the lines corresponding to an extra constructor produce an exhaustive matching. If any of these two checks fails, close the variant type.

(4) Convert all closed variant types to acceptor types. (*i.e.* remove the presence information.) Open variant types are not

p	::=	$\_$	wild card	τ	::=	α	variable
	$\mid$	$C^L\ p$	normal variant		$\mid$	$[C_i \text{ of } \tau_i]_1^n$	normal variant
	$\mid$	$C\ p$	poly. variant		$\mid$	$[\rangle\, C_i \text{ of } \tau_i]_1^n$	polymorphic variant
	$\mid$	$(p, \ldots, p)$	tuple		$\mid$	$\tau \times \cdots \times \tau$	tuple
	$\mid$	$p \mid p$	or-pattern		$\mid$	unit	unit tuple
L	::=	$\{C_1, \ldots, C_n\}$	constructors				

Fig. 1 Patterns and types

$$\textsc{Any}\quad \vdash \_ : \tau \qquad \textsc{Unit}\quad \vdash () : \mathsf{unit}$$

$$\textsc{Mono}\quad \frac{\vdash p : \tau}{\vdash C_k^L\ p : [C_i \text{ of } \tau_i]_1^n}\ 1 \le k \le n,\ L = \{C_i\}_1^n \qquad \textsc{Poly}\quad \frac{\vdash p : \tau}{\vdash C_k\ p : [\rangle\, C_i \text{ of } \tau_i]_1^n}\ 1 \le k \le n$$

$$\textsc{Tuple}\quad \frac{\vdash p_i : \tau_i \qquad (1 \le i \le n)}{\vdash (p_1, \ldots, p_n) : \tau_1 \times \cdots \times \tau_n} \qquad \textsc{Or}\quad \frac{\vdash p : \tau \quad \vdash p' : \tau}{\vdash p \mid p' : \tau}$$

Fig. 2 Typing rules

modified.

As you can see, all steps are symmetric. In particular, step 2 does not privilege a row or a column over another. Two different pattern-matching leading to the same matrix (modulo reordering of rows and columns) will result in the same typing.

Note also that the last step means that this algorithm is not monotonous: a presence type is not provably more general than an acceptor type (with the same constructors). Indeed, it cannot be unified with an acceptor type containing strictly less constructors. In general, being non-monotonous can break the principality of type inference. However, this algorithm only applies to the pattern part of the pattern-matching. As a result, it does not import any information from the environing expression, and is functionally computed from the patterns. This preserves the principality of type inference on programs.

5. Exhaustiveness and typing

In this section we formalize our typing strategy. We call it a strategy rather than an algorithm, because its goal is to give an intuitive definition of the typing obtained, rather than to pursue efficiency.

Since we are only interested in pattern-matching, we need just define patterns and their types, as done in figure 1. Patterns are either a wild card, a normal variant constructor C^L applied to a pattern, where L is the list of constructors for the corresponding sum type, a polymorphic variant constructor C applied to a pattern, a tuple (including the 0-ary unit tuple), or an or-pattern. Variables would be needed to bind values in expressions, but in the absence of expressions we can just replace them by wild cards. This also lets us consider a whole pattern-matching as a single or-pattern: they are equivalent for both typing and exhaustiveness. To further simplify the description, we used here the same notation for normal and polymorphic variants. This way we do not need to introduce explicit type definitions: a normal variant is just a variant for which the constructor list is fixed.

5.1 Rough typing

The corresponding typing rules are presented in figure 2. A pattern typing judgment is of them form $\vdash p : \tau$, where τ is the type of the pattern. Again, full typing would also provide a binding environment, but we don't need it in the absence of variables.

The first step of our typing algorithm is to infer the most general type for an or-pattern containing all the cases in a pattern-matching. That is, we want to infer τ such that for any τ', the judgment $\vdash p : \tau'$ is derivable if and only if there is a type substitution θ giving $\theta(\tau) = \tau'$. This is easily done by unification. Rather than detailing here the unification algorithm we refer you to other papers[2),4)]. Note that since we have only presence types for polymorphic variants in pattern types, unification is very simple, and Ohori's approach is sufficient[8)].

For instance consider the pattern

$$\begin{aligned} p = \ & T^{\{T,F\}}\ (), A\ (P^{\{P\}}\ ()) \\ \mid\ & T^{\{T,F\}}\ (), B\ (Q^{\{Q\}}\ ()) \\ \mid\ & F^{\{T,F\}}\ (), x \end{aligned}$$

its most general typing is (abbreviating unit):

$$\vdash p : [T \mid F] \times [\rangle\, A \text{ of } [P] \mid B \text{ of } [Q]]$$

$$\text{A-ANY}\quad \vdash M \sqsubseteq \_ : \tau$$

$$\text{A-MONO}\quad \frac{\vdash M \sqsubseteq p : \tau_k}{\vdash (C_k) \bullet B(\tau_1 \times \cdots \times \tau_{k-1}) \bullet M \bullet B(\tau_{k+1} \times \cdots \times \tau_n) \sqsubseteq C_k^L\ p : [C_i \text{ of } \tau_i]_{C_i \in L}}$$

$$\text{A-UNIT}\quad \vdash M \sqsubseteq () : \mathsf{unit}$$

$$\text{A-POLY}\quad \frac{\vdash M \sqsubseteq p : \tau_k}{\vdash (C_k) \bullet B(\tau_1 \times \cdots \times \tau_{k-1}) \bullet M \bullet B(\tau_{k+1} \times \cdots \times \tau_n) \sqsubseteq C_k\ p : [\rangle\, C_i \text{ of } \tau_i]_1^n}$$

$$\text{A-TUPLE}\quad \frac{\vdash M_i \sqsubseteq p_i : \tau_i \quad (1 \le i \le n)}{\vdash M_1 \bullet \ldots \bullet M_n \sqsubseteq (p_1, \ldots, p_n) : \tau_1 \times \cdots \times \tau_n}$$

$$\text{A-OR1}\quad \frac{\vdash M \sqsubseteq p_1 : \tau}{\vdash M \sqsubseteq p_1 \mid p_2 : \tau}$$

$$\text{A-OR2}\quad \frac{\vdash M \sqsubseteq p_2 : \tau}{\vdash M \sqsubseteq p_1 \mid p_2 : \tau}$$

Fig. 3 Acceptability rules

5.2 Exhaustiveness matrix

The next step is to build the exhaustiveness matrix corresponding to the inferred type. This is done by induction on the structure of the type.

$$M(\alpha) = ()$$

$$M(\mathsf{unit}) = ()$$

$$M([C_i^L \text{ of } \tau_i]_1^n) = \begin{pmatrix} C_1 \bullet M(\tau_1) \bullet B(\tau_2 \times \cdots \times \tau_n) \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ C_n \bullet B(\tau_1 \times \cdots \times \tau_{n-1}) \bullet M(\tau_n) \end{pmatrix}$$

$$M([\rangle\, C_i \text{ of } \tau_i]_1^n) = \begin{pmatrix} C_1 \bullet M(\tau_1) \bullet B(\tau_2 \times \cdots \times \tau_n) \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ C_n \bullet B(\tau_1 \times \cdots \times \tau_{n-1}) \bullet M(\tau_n) \\ C_+ \bullet B(\tau_1 \times \cdots \times \tau_n) \end{pmatrix}$$

$$M(\tau_1 \times \cdots \times \tau_n) = M(\tau_1) \bullet \ldots \bullet M(\tau_n)$$

The matrix contains variant constructors C, the extra constructor C_+, and blank marks $\top$. In this definition, $()$ denotes a matrix of 1 line and 0 column; $M \bullet N$ denotes matrix line-wise concatenation; and $B(\tau)$ is a 1-line matrix of $\top$'s with the same number of columns as $M(\tau)$.

By line-wise concatenation we mean the following operation: assuming that $M = (m_{ij})$ is a l-line l'-column matrix and $N = (n_{ij})$ is a k-line k'-column matrix, then $P = (p_{ij})$ is a $l \times k$-line $l' + k'$-column matrix and

$$p_{ij} = \begin{cases} m_{((i-1)/k+1)j} & j \le l' \\ m_{((i-1) \bmod k+1)(j-l')} & j > l'. \end{cases}$$

Applying the above definition, the exhaustiveness matrix for p is:

$$M([T \mid F] \times [\rangle\, A \text{ of } [P] \mid B \text{ of } [Q]]) = \begin{pmatrix} T & A & P & \top \\ T & B & \top & Q \\ T & C_+ & \top & \top \\ F & A & P & \top \\ F & B & \top & Q \\ F & C_+ & \top & \top \end{pmatrix}$$

As you can see the exhaustiveness matrix has a size exponential in the size of the type. A practical algorithm would not build it physically, but only enumerate it as needed.

Note that by construction, each of its columns corresponds to a single variant type. We note this type $t_j(M)$. In this example,

$$t_1(M(\tau)) = [T \mid F]$$
$$t_2(M(\tau)) = [\rangle\, A \text{ of } [P] \mid B \text{ of } [Q]]$$
$$t_3(M(\tau)) = [P]$$
$$t_4(M(\tau)) = [Q]$$

A pattern-matching is exhaustive when all lines in the exhaustiveness matrix are accepted by a pattern. For this we define the acceptability relation at a type τ in figure 3. If $(m_{i1} \ldots m_{in})$ is the i^{th} line of $M(\tau)$, then it is accepted by p if and only if $\vdash (m_{i1} \ldots m_{in}) \sqsubseteq p : \tau$ is derivable using the acceptability rules.

One can easily see that in our example, the 3rd line of the exhaustiveness matrix is not accepted. This means that, using directly the principal type given by the typing rules, this pattern-matching would not be exhaustive.

Note that the order of clauses inside an or-pattern, or the order of lines inside the exhaustiveness matrix does not change acceptability. Moreover, if τ is a tuple type, changing the order of its components creates a matrix identical to the original one modulo permutations of lines and columns, and acceptability of each line does not change (if we permute the patterns identically.) From these two facts we can see that the exhaustiveness of a pattern-matching at a given type is a symmetrical property, preserved by both permutation of or-patterns and tuple-patterns.

5.3 Type refinement

The third step of our algorithm is to use exhaustiveness check to decide which polymorphic variant types should be kept open as presence types, and which should be reverted to closed

acceptor types. We proceed in the following way. For each column corresponding to a polymorphic variant type, we extract all the lines with only a C_+ in this column (but not in other ones). If all these lines are accepted, then the type is kept open, otherwise it must be closed. Formally, assuming $M(\tau) = (m_{ij})$ of width n, then for all k's such that

$$(\exists i) \begin{cases} \vdash (m_{i1} \dots m_{in}) \not\sqsubseteq p : \tau \\ (\forall j)\ m_{ij} = C_+ \Leftrightarrow j = k \end{cases}$$

the polymorphic variant type $t_k(M(\tau)) = [\rangle\, C_i \text{ of } \tau_i]_1^l$ corresponding to the column k must be converted to the acceptor type $[\langle\, C_i \text{ of } \tau_i]_1^l$. Since permuting or-patterns or tuple-patterns would only change the order of lines and columns in the exhaustiveness matrix, the choice of the types to close is symmetrical.

The type judgment $\vdash_{\mathit{fin}} p : \tau_{\mathit{fin}}$ indicates that τ_{fin} is the final type obtained for p by this whole process. That is, the principal type τ of p was refined into τ_{fin} by checking the exhaustiveness matrix $M(\tau)$.

Going on with our running example, the relevant lines are:

$$\begin{pmatrix} T & C_+ & \top & \top \\ F & C_+ & \top & \top \end{pmatrix}$$

While the second line is accepted

$$\vdash (F\ C_+\ \top\ \top) \sqsubseteq F\ (), \_ : \tau$$

since $\_$ matches anything, the first line is not —only the A and B cases are handled by p. As a result, the corresponding variant type must be closed, and the final pattern type is:

$$\vdash_{\mathit{fin}} p : [T \mid F] \times [\langle\, A \text{ of } [P] \mid B \text{ of } [Q]]$$

If we compute the exhaustiveness matrix corresponding to this new type, it contains no longer C_+ lines (an acceptor type $[\langle \dots]$ produces the same matrix as a normal type $[\dots]$)

$$M([T \mid F] \times [\langle\, A \text{ of } [P] \mid B \text{ of } [Q]]) = \begin{pmatrix} T & A & P & \top \\ T & B & \top & Q \\ F & A & P & \top \\ F & B & \top & Q \end{pmatrix}$$

This means that with this new type our pattern-matching is exhaustive.

Let us now check with another example, namely a simplification of the function `h` in the previous section.

$$\vdash A\ (), \_ \mid \_, B\ () : [\rangle\, A] \times [\rangle\, B]$$

The corresponding matrix is:

$$M([\rangle A] \times [\rangle B]) = \begin{pmatrix} A & B \\ A & C_+ \\ C_+ & B \\ C_+ & C_+ \end{pmatrix}$$

For the first column, two lines contain C_+, but only one is relevant, as relevant lines must contain only one C_+. It is accepted.

$$\vdash (C_+\ B) \sqsubseteq \_, B : [\rangle\, A] \times [\rangle\, B]$$

Similarly, the only relevant line for the second column is also accepted.

$$\vdash (A\ C_+) \sqsubseteq A, \_ : [\rangle\, A] \times [\rangle\, B]$$

As a result, the two variant types must be kept open.

$$\vdash_{\mathit{fin}} A\ (), \_ \mid \_, B\ () : [\rangle\, A] \times [\rangle\, B]$$

Of course, since the last line is not accepted

$$\vdash (C_+\ C_+) \not\sqsubseteq A\ (), \_ \mid \_, B\ () : [\rangle\, A] \times [\rangle\, B]$$

this also means that the resulting pattern-matching is not exhaustive.

6. Conclusion

We have seen in this paper a possible typing strategy for deep pattern-matching containing polymorphic variants. This strategy attempts at making the pattern-matching exhaustive by eventually closing some variant types. The behavior is symmetrical with respect to or-pattern and tuple-pattern order.

Whether this is the best strategy, and the inferred type is always the most intuitive one, remains an open question.

There are two possible criticisms. On one side, contrary to some strategies used in older versions of Objective Caml, this strategy does not guarantee that all polymorphic variant pattern-matchings are complete, as we have seen in our last example. This would be a nice property to have, but it seems more coherent with other datatypes to limit ourselves to a warning, when enforcing it would require an overly restrictive type.

On the other side, the type produced may sometimes be more restrictive than the one intended by the programmer. This could be the case when the type causes a pattern to be unused.

```
let f = function
  | _, `A -> 1
  | true, _ -> 2
Warning: this pattern is unused.
val f : bool * [ `A ] -> int
```

One could require the strategy to force all patterns to be used. However, this cannot be enforced, as this warning can appear even in pattern-matchings without variants. And we believe that, as this definition is inherently ambiguous (at the type level), a warning is the right behavior to obtain in this case.

References

1) Augustsson, L.: Compiling Pattern Matching, *Proc. ACM Symposium on Functional Programming and Computer Architectures* (Jouannaud, J.-P.(ed.)), Springer-Verlag LNCS 201, pp. 368–381 (1985).

2) Garrigue, J.: Programming with Polymorphic Variants, *ML Workshop*, Baltimore (1998).

3) Garrigue, J.: Code reuse through polymorphic variants, *Workshop on Foundations of Software Engineering*, Lecture Notes in Software Science, No. 25, Sasaguri, Japan, Kindai Kagakusha, pp. 93–100 (2000).

4) Garrigue, J.: Simple type inference for structural polymorphism, *The Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon (2002).

5) Leroy, X., Doligez, D., Garrigue, J., Rémy, D. and Vouillon, J.: *The Objective Caml system release 3.07, Documentation and user's manual*, Projet Cristal, INRIA (2003).

6) Maranget, L.: Les avertissements du filtrage, *Journées Francophones des Langages Applicatifs* (2003).

7) Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts (1990).

8) Ohori, A.: A Polymorphic Record Calculus and Its Compilation, *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 6, pp. 844–895 (1995).

9) Wadler, P.: Efficient Compilation of Pattern-Matching, *The Implementation of Functional Programming Languages* (Peyton Jones(ed.)), Prentice-Hall, chapter 5, pp. 78–103 (1987).