Ambiguous pattern variables

Gabriel Scherer, Luc Maranget, Thomas Réfis

Ambiguous pattern variables

Gabriel Scherer, Luc Maranget, Thomas Réfis

June 15, 2016

The or-pattern `(p | q)` matches a value v if either p or q match v. It may happen that both p and q match certain values, but that they don't bind their variables at the same places. OCaml specifies that the left pattern p then takes precedence, but users intuitively expect an angelistic, making the "best" choice. Subtle bugs arise from this mismatch. When are `(p | q)` and `(q | p)` observably different?

To correctly answer this question we had to go back to pattern matrices, the primary technique to compile patterns and analyse them for exhaustivity, redundant clauses, etc. There is a generational gap: pattern matching was actively studied when most ML languages were first implemented, but many of today students' and practitioners trust our elders to maintain and improve them. Read on for your decadely fix of pattern matching theory!

A bad surprise Consider the following OCaml matching clause:

```
| (Const n, a) | (a, Const n)
    when is_neutral n -> a
```

This clause, part of a simplification function on some symbolic monoid expressions, uses two interesting features of OCaml pattern matching: `when` guards and or-patterns.

A clause of the form p `when` g `->` e matches a pattern scrutinee if the pattern p matches, and the guard g, an expression of type `bool`, evaluates to `true` in the environment enriched with the variables bound in p. Guards occur at the clause level, they cannot occur deep inside a pattern.

The semantics of our above example seems clear: when given a pair whose left or right element is of the form `Some n`, where `n` is neutral, it matches and returns the other element of the pair.

Unfortunately, this code contains a subtle bug: when passed an input of the form `(Const` v`, Const` n`)` where v is not neutral but n is, the clause does *not* match! This goes against our natural intuition of what the code means, but it is easily explained by the OCaml semantics detailed above. A guarded clause p `when` g `->` e matches the scrutinee against p first, and checks g second. Our input matches both sides of the or-pattern; by the specified left-to-right order, the captured environment binds the pattern variable `n` to the value v (not n). The test `is_neutral n` fails in this environment, so the clause does not match the scrutinee.

A new warning This is not an *implementation* bug, the behavior is as specified. This is a *usability* bug, as our intuition contradicts the specification.

There is no easy way to change the semantics to match user expectations. The intuitive semantics of "try both branches" does not extend gracefully to or-patterns that are in depth rather that at the toplevel of the pattern. Another approach would be to allow `when` guards in depth inside patterns, but that would be a very invasive change, going against the current design stance of remaining in the pattern fragment that is easy to compile – and correspondingly has excellent exhaustiveness and usefulness warnings. The last resort, then, is to at least complain about it: detect this unfortunate situation and warn the user that the behavior may not be the intended one. The mission statement for this new warning was as follows: "warn on `(`p_1 `|` q_2`)` `when` g when an input could pass the guard g when matched by p_2, and fail when matched by p_1.

This new warning was included in OCaml 4.03, released in April 2016.

Specification and non-examples A pattern p may or may not match a value v, but if it contains or-patterns it may match it in several different ways. Let us define `matches`(p, v) as the ordered list of matching environments, binding the free variables of p to sub-parts of v; if it is the empty list, then the pattern does not match the value.

A variable `x` $\in p$ is *ambiguous* if there exists a value v such that distinct environments of `matches`(p, v) map `x` to distinct values. We must warn on guarded clauses

(`p when g -> e`) when the guard g uses an ambiguous variable of p.

In the case of `((x, None, _) | (x, _, None))`, the variable `x` is not ambiguous, as it will always bind the same sub-value for any input.

In the case of `((x, None, _) | (_, Some _, x))`, the variable `x` is not ambiguous, as there is no input value that may match both sides of the pattern.

Implementation Pattern matrices are a common representation for pattern-matching algorithms. A $m \times n$ pattern matrix corresponds to a m-disjunction of pattern on n arguments matched in parallel:

$$\begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{bmatrix} \quad \text{is} \quad \begin{array}{l} \mid (p_{1,1}, p_{1,2}, \cdots, p_{1,n}) \\ \mid (p_{2,1}, p_{2,2}, \cdots, p_{2,n}) \\ \mid \cdots \\ \mid (p_{m,1}, p_{m,2}, \cdots, p_{m,n}) \end{array}$$

A central operation is to split a matrix into sub-matrices along a given column, for example the first column. Consider the matrix

$$\begin{bmatrix} \mathtt{K}_1(q_{1,1}) & p_{1,2} & \cdots & p_{1,n} \\ \mathtt{K}_2(q_{2,1}, q_{2,2}) & p_{2,2} & \cdots & p_{2,n} \\ \_ & p_{3,1} & \cdots & p_{3,n} \\ \mathtt{K}_2(q_{4,1}, q_{4,2}) & p_{4,2} & \cdots & p_{4,n} \end{bmatrix}$$

We know that a n-tuple of values matching some row in this sub-matrix has a first value that either starts with the head constructor $\mathtt{K}_1$, or $\mathtt{K}_2$, or another one. This corresponds to studying the following sub-matrices, that describe the shape of all possible values matching this pattern – with the head constructor of the first column removed:

$$\begin{bmatrix} q_1 & p_{1,2} & \cdots & p_{1,n} \\ \_ & p_{3,1} & \cdots & p_{3,n} \end{bmatrix} \qquad \begin{vmatrix} q_{2_1} & q_{2,2} & p_{1,2} & \cdots & p_{1,n} \\ \_ & \_ & p_{3,1} & \cdots & p_{3,n} \\ q_{4_1} & q_{4,2} & p_{4,2} & \cdots & p_{4,n} \end{vmatrix}$$

$$\begin{bmatrix} \_ & p_{3,1} & \cdots & p_{3,n} \end{bmatrix}$$

If a pattern in the column we wish to split does not start with a head constructor or `_`, but with an or-pattern, one can simplify it into two rows:

$$\begin{bmatrix} (q_1 \mid q_2) & r_1 \\ \vdots & \ddots \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} q_1 & r \\ q_2 & r \\ \vdots & \ddots \end{bmatrix}$$

Finally, before splitting on a column, one may consider the variables that are bound at the head of the patterns of this row. Consider our previous examples enriched with variable bindings (the pattern p `as x` matches the scrutinee against p and also binds it to the variable `x`):

$$\begin{bmatrix} \mathtt{K}_1(q_{1,1}) \text{ as x} & p_{1,2} & \cdots & p_{1,n} \\ \mathtt{K}_2(q_{2,1}, q_{2,2}) \text{ as x as y} & p_{2,2} & \cdots & p_{2,n} \\ \mathtt{x} & p_{3,1} & \cdots & p_{3,n} \\ \mathtt{K}_2(q_{4,1}, q_{4,2}) \text{ as x} & p_{4,2} & \cdots & p_{4,n} \end{bmatrix}$$

The variable `x` is bound at the head of each pattern of the first column, so it is a stable variable: for any matching n-tuple of values $(v_1, v_2, \ldots, v_n)$, it will be bound to the same sub-value v_1. On the contrary, `y` is bound at the head of the second row but no others, it is an ambiguous variables. We know that all rows bind the same environment, so `y` appears in other places in other rows; we know that each variable occurs only once in each row, so `y` is not stable in another column.

Our implementation repeatedly peels of the variables bound at the head of the first column; it computes the stable variables for this column as the intersection of the variables appearing in all rows. It then splits the matrix into sub-matrices, and recursively computes stable variables for each sub-matrix. A variable is stable for the whole matrix if it is stable for the first row, or stable for all submatrices.

Correctness In absence of pattern variables, pattern matrices and splitting can be presented as an implementation device to compute, for a given pattern p, a *covering* $\sum_{i \in I} s_i$, which is a family of *simple* patterns (no or-patterns inside), such that p is equivalent to the disjunction $(s_1 \mid s_2 \mid \ldots \mid s_n)$, and such that two distinct s_i, $s_{i'}$ are *disjoint*, no value is matched by both.

Once we take pattern variable into accounts, two simple patterns may match the same values (hopelessly non-disjoint) but bind variables different. A covering of p is then an equivalent pattern p' (such that $\forall v, \mathtt{matches}(p, v) = \mathtt{matches}(p', v)$) of the form $\sum_{i \in I} \sum_{j \in J_i} s_{i,j}$, where two $s_{i,j}$, $s_{i',j'}$ are disjoint whenever $i \neq i'$, but two $s_{i,j}$, $s_{i,j'}$ for the same $i \in I$ match the same values – they may export distinct environments. A variable is *stable* in a sub-group $\sum_j s_{i,j}$ if it is bound at the same position in each simple pattern; the stable variables of p are the intersections of the stable variables of each group.

Acknowledgments This subtle bug was brought to our attention by Arthur Charguéraud, Martin Clochard and Claude Marché. François Pottier made the elegant remark that ambiguous variables correspond to non-commutative or-patterns – $(p \mid q)$ different from $(q \mid p)$.