Ambiguous pattern variables

Gabriel Scherer, Luc Maranget, Thomas Réfis

July 29, 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 angelic behavior, 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 analyze 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's 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 `Const 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".

We introduced this new warning 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, and *stable* otherwise. We must warn when a guard uses an ambiguous variable.

`x` is stable in `((x, None, _) | (x, _, None))`, as it will always bind the same sub-value for any input.

`x` is stable in `((x, None, _) | (_, Some _, x))`, as no value may match both sides of the or-pattern.

Pattern matrices 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:

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

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

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

The first element of a n-tuple matching some row of the matrix starts with either (1) the head constructor $\mathtt{K}_1$, or (2) $\mathtt{K}_2$, or (3) another one. The three following sub-matrices thus describe the shape of all possible values matching this pattern – with the head constructor of the first column removed:

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

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

A variable is stable in a matrix if it is stable in each of its sub-matrices.

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:

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

After repeated splitting, a column ends up with only nullary constructors or universal patterns `_`; the next split removes the column. Eventually, repeated splitting terminates on a matrix with several rows but no columns.

Binding sets When splitting a matrix into sub-matrices, we peel off a layer of head constructors, and thus lose information on any variable bound at this position in the patterns.

To correctly compute stable variables, we need to keep track of these binding sites: we enrich pattern matrices with information on what variables were peeled off each row. Our matrices are now of the form

$$\left[\begin{array}{ccc|ccc} B_{1,1} & \ldots & B_{1,l} & p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ B_{m,1} & \ldots & B_{m,l} & p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{array}\right]$$

where the $B_{i,k}$ are *binding sets*, sets of variables found in the same position during pattern traversal. Variables of different columns correspond to different binding positions, so they may bind distinct values.

The type-checker ensures that the two sides of an or-pattern `(`p `|` q`)` bind the same variables, and that patterns are otherwise linear – each variable occurs once. This guarantees that all rows bind the same environment, and that each variable occurs either in a single pattern of the row, or in one of the binding sets.

Variable binding at the head of the leftmost pattern are moved it to the rightmost binding set.

$$\left[\ldots\ B_{1,l} \mid \texttt{(}p\ \texttt{as x)}\ \ldots\right] \Rightarrow \left[\ldots\ (B_{1,l} \cup \{x\}) \mid p\ \ldots\right]$$
$$\left[\ldots\ B_{1,l} \mid \texttt{x}\ \ldots\right] \Rightarrow \left[\ldots\ (B_{1,l} \cup \{x\}) \mid \_\ \ldots\right]$$

We insert a new binding set when splitting on the head constructor of the first pattern row: head variables of the new rows bind to a different position.

$$\begin{array}{c} \left[B_{i,1} \quad \ldots \quad B_{i,l} \quad \mid \quad \mathtt{K}(q_1, \ldots, q_k) \quad p_{i,2} \quad \ldots \quad p_{i,m}\right] \\ \Rightarrow \left[B_{i,1} \quad \ldots \quad B_{i,l} \quad \emptyset \quad \mid \quad q_1 \quad \ldots \quad q_k \quad p_{i,2} \quad \ldots \quad p_{i,m}\right] \end{array}$$

When traversal ends on a matrix with empty rows, we compute stability of this matrix from the binding sets:

$$\left[\begin{array}{ccc|} B_{1,1} & \ldots & B_{1,l} & \mid \\ \vdots & \vdots & \vdots & \mid \\ B_{m,1} & \ldots & B_{m,l} & \mid \end{array}\right]$$

Binding sets along a given column correspond to variables that are bound at the same position for all possible ways to enter this sub-matrix. The intersection of these sets thus gives the stable variables of the column. Because the variable sets are disjoint, a variable stable for a column cannot appear anywhere else.

Acknowledgments This subtle bug was brought to our attention by Arthur Charguéraud, Martin Clochard and Claude Marché. François Pottier made the elegant remark that ambiguity corresponds to non-commutative or-patterns – `(`p `|` q`)` different from `(`q `|` p`)`.