We thank the reviewers for their feedback, many suggestions for improving the presentation.

Reviewer 2 laments (echoed by reviewer 3) that "the paper does not describe the API of the pattern-matching library, but only the low-level mechanisms, like type testing. This makes it hard to get the big picture regarding exactly what has been achieved"

Our aim was to produce a viable library implementation for FP-style pattern matching in C++ for scientific experimentation and evaluation. To succeed, we must simultaneously meet a number of challenges: provide a suitable notation and efficient implementation of a type switch that is open to class extensions, matches OCaml in performance, and beats the visitor pattern. These are not sufficient by themselves, but when done simultaneously and on variety of industrial platforms – present a significant contribution. Without them, we risk the library not be adopted by users. This paper reports on the "system" aspect of this challenge.

The language and library design issues for our pattern matching is a subject of a separate paper, currently under revision for ECOOP12. To avoid sidetracking and dependence on that paper, we shall present our example in the form that won’t require any definitions from the user:  
 Case(Plus) return eval(matched.e1)+eval(matched.e2); etc.   
We shall also remove any other distracting information not discussed in the paper.

We are familiar with the work suggested by the reviewers (some is referenced in TR). Our focus is on a portable library solution with special attention to type switching (not type testing) and open class extensions. We prefer not to limit presentation to open case, because on one side it underperforms in comparison to closed solution, while on the other, closed ADTs are important in practice for case analysis, exhaustiveness checking and providing better diagnostics. We shall limit its discussion to presentation of performance numbers and comparison to open approach.

The conditions we identified for our approach to work are very general and work with the Microsoft’s and other C++ compilers. Multi-threading is not problematic, we just haven’t had time to experiment which of the many obvious approaches works best. We will list a few.

Our end-to-end benchmark implements a C++ pretty-printer with visitors and type-switching approaches. When tested on headers from the standard library, we see a speed-up of up to 33% for files over 500 lines of pre-processed code and an average slowdown of 9% for smaller ones, where our startup-cost dominates. We shall expand this comparison and provide more details about our adoptable caching mechanism; meanwhile we uploaded some memory and timing measurements here: https://parasol.tamu.edu/~yuriys/pm/

Dynamic languages have been optimizing method lookup for decades, but they are addressing a different problem. We are optimizing a double dispatch that is already optimized to two array lookups.

Making the paper more accessible by reducing its scope could easily compromise our claim of a comprehensive solution. Making the paper more accessible by adding tutorial material conflicts with the page limit. We will of course try both guided by the reviewer comments.