We would like to thank the reviewers for the useful feedback. We will integrate it into the final version along with discussion of suggested references, which we missed but find very relevant.

The main goal of our approach to pattern matching was not presenting a solution that we would like to standardize as a library, but a solution that would let us experiment with a multitude of design choices in order to make reasonable and informed decisions about possible language solution. Certain usability issues like odd names, syntax or poor error messages will go away in such solution.

We generally prefer inlining and type-safe composition offered by templates to preprocessor, which we use only when it gives us significant notational advantage (KS, KV, etc.) or when we need access to the lexical scope (Match, Qua etc.). Different macros are used because of slight semantic differences and thus difference in the generated syntactic structure to support it (e.g. Case vs. Qua). We shall explain this better in the paper.

Our use of natural semantics to present the semantics of our extension was motivated by the fact that template facilities of C++ form a functional compile-time sublanguage, and inference rules mimic almost exactly the actual implementation through expression templates. We shall add an example of how to map a particular rule into actual C++, which will make that correspondence more vivid. The same applies to typing rules, which represent almost exactly the type composition performed by the compiler to type patterns and lazy expressions introduced by our library.

Performance is indeed a differentiating factor of our solution in comparison to library solutions in other languages, however it is a consequence of a larger observation. Pattern-matching libraries in other languages achieve their composability at run-time, effectively degrading performance because of dynamic dispatch. Our library achieves composability at compile-time allowing the compiler to inline the entire matching expression, with the only exception of dynamic type tests required sometime by constructor patterns. Similar to other library solution, our patterns are first-class citizens because we can independently create them, store in variables, compose and pass to functions – they do not have to be immediately applied to the subject, as argued by the third reviewer.

We thank the reviewers for their thoughtful feedback. We will integrate their suggestions (where we agree) in the final version, including discussions of suggested related work we missed.

The primary goal of this research work is the design and evaluation of a library-based solution to the problem of integrating a pattern matching facility into an existing mainstream programming language supporting object-oriented programming and parameterized polymorphism. It is clear that as a library solution may have syntactic issues that a language extension may lack. However, one of the key contributions of this work is the design of a solution that is both elegant and competitive in efficiency with other well established purely language extension solutions. This approach follows the principle that language design is library design, and allows us to experiment with usability issues and to study scalability. The main ingredients are standard C++ templates facilities, coupled with inlining, with some syntactic support based on a couple of macros. It is clear that a pure language extension will obviate the need for macros.

The overlap between the PLDI submission and this submission is very minimal. The PLDI submission (which focused on the system-level implementation details) was unfortunately rejected. So, we can incorporate motivation, more implementation material, and performance evaluation from that paper (within space limits).

One reviewer suggests that the results are underwhelming because they do not generalize. We do not agree with this characterization, for the solutions are quite general in both principle and practice as evidenced the algorithm discussed in the paper and by practical tests, we conducted. The "considerable syntactic overhead" predicted by the reviewer do not materialize in practice; on the contrary we found that our solution requires less clutter than the remarkable work of Joost Visser cited in the feedback.

To clarify some misunderstandings, xi is the Lazy Expression category described in the grammar, while typing rules only give types to terms falling into Extended Pattern (omega) category. Chi is indeed a variable, we shall put Sigma back instead of our notation for contained value to avoid confusion. Pattern application may return a value convertible to bool (pointer to derived class in our case), but we shall omit that technicality for clarity sake. We motivate in 4.2 why the choice of semantics for generalized n+k patterns can be domain specific and indeed leave the choice to the user. We do support nested patterns, including any combinations of pattern kinds described in the grammar.