Our aim was to produce a viable implementation for FP-style pattern matching for C++. To succeed, we must \*simultaneously\* meet a number of challenges. Providing a notation that works in the context of C++, providing a type switch that match OCaml in performance, and providing a type switch that beats the visitor pattern in C++ and Java are not by themselves sufficient, we must do all simultaneously and on a variety of industrial platforms. Whatever we do, doing so simultaneously is a significant contribution.

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.

The Microsoft C++ implementation meets our assumptions, as does every other C++ implementation.

Multi-threading is not problematic, we just haven’t had time to see which of the many obvious approaches works best.

Simple type switches require no added effort from the programmer; to use introduce variable names in cases, as in the introductory example takes two lines of code; we’ll add them.

We will move the explanation of the difference between type testing, type identification, and type switching from the TR.

The language design issues for pattern matching for C++ is in a different paper. We don’t have space for both that and the implementation. We can add a bit at the cost of some implementation details.

All memorization schemes are somewhat similar. Our scheme ??? <<here we \*must\* briefly state what is our contribution>>. Many approaches are closed (we treat the closed case as an optimization) and several report cost above those of visitor pattern use (which is unacceptable to us).

No, we cannot restrict ourselves to the open/OO case; our aim is to equal or beat the gold standard in the areas we address, and OCaml is the gold standard for many closed cases.

Our benchmarks are not (all) repeatedly hitting the same receiver, but we can quote figures showing that real applications speed up when using our pattern matching (against optimized industrial implementations of C++) <<do you have some numbers?; simple is sufficient>>

Dynamic languages have of course 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 table lookups.

We thank the reviewers for the many useful suggestions for improving the presentation.