Our aim is to produce an efficient type switch for C++. By efficient we mean comparable or faster than golden standard or alternatives, otherwise the feature may not be adopted. OCaml and tags-switch are such for closed ADT-like structures; type-testing + decision-trees is NOT such for OPEN hierarchical datatypes, because they are comparable to visitors for about 4 clauses (with overhead of an integer division per type-test). We prefer not to limit presentation to open solution because it underperforms in comparison to closed solution. We shall limit discussion of closed case to presentation of performance numbers and comparison to open approach.

The language and library design issues for pattern matching is a subject of a separate paper currently under revision for ECOOP12. To avoid side-tracking 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 reference to multi-methods, which became a distraction. The paragraph on difference between type-testing, type-identification and type-switching fell out, resulting in side-tracking into type-testing techniques. We shall move the explanation back from the TR. We shall also put back mentioning of Yossi Gil’s and others work on type-testing, even though type-switching based on type-testing is not efficient in our terms (section 2.2).

Multi-threading is not problematic, we just haven’t had time to experiment which of the many obvious approaches works best. We shall surely list a few. The Microsoft C++ implementation meets our assumptions, as does most C++ implementations, the phrasing became misleading.

We shall cite Holze’s work to introduce call-site-caching techniques. Our approach, however, needs neither re-compilation, nor re-linking nor even any computations at dynamic-linking time. We map arbitrary number of receiver-types to \*fixed\* number of targets, which lets us generate in a library code layout that can be executed sequentially and directly (re-establishing invariants guaranteed by sequential execution). Our benchmarks are not (all) repeatedly hitting the same receiver, only the repetitive benchmark does. Note that repetitive benchmark compares our performance to performance of double-dispatch essentially optimized with Inline Caching by call-target prediction, making visitors twice faster than usual! This is why our repetitive numbers for open solution are at disadvantage in comparison to sequential and random.

We can cite Chambers and Chen instead of earlier Ernst,Kaplan,Chambers as their discussion of multiway-type-testing stresses our point – it is efficient when Class IDs are sequential and closed, while they have to resort to decision trees otherwise.

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.

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.

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