A C++ Transformation Framework

12th Nordic Workshop on Programming Theory,
Bergen, October 2000
http://www.ii.uib.no/~nwpt00

Otto Skrove Bagge[1], Magne Haveraaen[1], and Eelco Visser[2]

[1] Institutt for Informatikk, Universitetet i Bergen,
N-5020 Bergen, Norway,
(otto|magne)@ii.uib.no
[2] Instituut voor Informatica en Informatiekunde, Universiteit Utrecht,
P.O.Box 80089, NL-3508 TB Utrecht, The Netherlands,
visser@cs.uu.nl

CodeBoost is a tool for source-to-source transformation and optimisation of C++ programs. It is intended to be used as a test-bed for various high-level optimisations; the traditional textbook optimisations are assumed to be handled by the C++ compiler. The CodeBoost optimiser will attempt to bridge the gap between a human-friendly coding style and current compiler/optimiser implementations. The C++ transformation framework may be useful for other projects as well—it has already been used in an experimental instrumentation and profiling tool.

CodeBoost is part of the Sophus project[3, 2], which explores the use of high-level abstractions for numerical applications. This has great advantages in terms of easier programming and maintenance, but slower performance has hindered the adoption of these techniques in HPC. Current compilers have proven unable to optimise sufficiently, partly due to low demand for such optimisations, and partly because some of the most effective optimisations are in conflict with the C++ standard[4, 1].

The CodeBoost optimiser gives the user control over the optimisation process. New optimisations can be added easily, and optimisations can be applied (and re-applied) in any order. Because the output is readable C++ code, it is easy to see the effects of the transformations. Three different optimisations are currently on the way, and more are planned for the future.

The first optimisation is motivated by the mathematical notation style used in Sophus, which greatly improves the readability of complex expressions. For instance, we write `A = A + B * c`, instead of something like `mul(B, c, t); add(A, t, A);` (possibly with the function bodies hand-inlined). Unfortunately, this programmer-friendly style is bad for performance; the result must be copied, and each operator call involves the creation of a new temporary. This might be fine with only a few bytes of data, but unacceptable when working with many megabytes. If the optimiser knows that `A += B` means the same as `A = A + B`, it can rewrite the expression to use only one temporary, and eliminate copying of the result. This is what the C++ compiler does for built-in types, but it can't do it for user-defined types, because the standard forbids assuming any relation between `+` and `+=`. If we demand that this relation holds, or allow the user to specify it for each case, we can perform the optimisation.

But even if we do this optimisation, we are left with another performance problem; the operators are typically defined in terms of loops, thus the computation is done in several passes over the data. This causes unnecessary cache misses and index computations. A smart optimiser can inline the operators, and use loop fusion to fold consecutive loops into a single loop. Your compiler might actually do loop fusion in some cases, but most likely it won't look for this case, and miss the opportunity.

Inlining a function eliminates the function-call overhead, and allows for further optimisations—such as loop fusion. C++ provides the `inline` keyword as a hint to the compiler that a function can be inlined, but this is merely a hint; the compiler is free to ignore it. Depending on the compiler, seemingly easy-to-inline code may run a lot slower than hand-inlined code. With proper inlining, we can use functions actively, and get better readability, maintainability and flexibility.

The CodeBoost framework is limited to processing a simplified version of the C++ language. Many irrelevant features, such as namespaces, pointers to functions, and inheritance, have been ignored to keep the implementation concise and manageable. New features are added as the need arises. The restrictions are motivated not only by implementation concerns, but also by good coding practice and assumptions made by the CodeBoost optimiser.

The implementation is based on standard compiler design, with clearly separate parts for each step in the process. A modified version of OpenC++ is used for parsing. All other components are written in Stratego [6]. Stratego is a language for specifying program transformations based on the paradigm of rewriting strategies. Transformations are described by a combination of concise rewrite rules and strategies allowing precise control over the order and position of application of rules. This makes transformation rules independent of the overall transformation they are applied in and reusable in other transformations. Strategies can be parameterised with rules to apply and thus make it possible to develop libraries of reusable transformation strategies.

The CodeBoost C++ transformation framework provides a library of standard transformation rules and strategies on C++ programs. Intermediate output from CodeBoost is in the ATerm format [5], and is primarily useful for Stratego programs, but is also accessible from C, C++ and Java, and is easily parsable. The final output is readable C++ code.

References

1. Programming languages — C++. International Standard ISO/IEC 14882, ISO/IEC/ANSI, 1998.
2. T.B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*.
3. Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.
4. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.
5. M. G. J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
6. Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).