Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs[*]

Otto Skrove Bagge
Chr. Michelsen Research AS
otto@codeboost.org

Karl Trygve Kalleberg
Department of Informatics
University of Bergen
karltk@ii.uib.no

Magne Haveraaen
Department of Informatics
University of Bergen
magne@ii.uib.no

Eelco Visser
Institute of Information and Computing Sciences
Utrecht University
visser@cs.uu.nl

Abstract

The use of a high-level, abstract coding style can greatly increase developer productivity. For numerical software, this can result in drastically reduced run-time performance. High-level, domain-specific optimisations can eliminate much of the overhead caused by an abstract coding style, but current compilers have poor support for domain-specific optimisation.

In this paper we present CodeBoost, a source-to-source transformation tool for domain-specific optimisation of C++ programs. CodeBoost performs parsing, semantic analysis and pretty-printing, and transformations can be implemented either in the Stratego program transformation language, or as user-defined rewrite rules embedded within the C++ program. CodeBoost has been used with great success to optimise numerical applications written in the Sophus high-level coding style.

We discuss the overall design of the CodeBoost transformation framework, and take a closer look at two important features of CodeBoost: user-defined rules and totem annotations. We also show briefly how CodeBoost is used to optimise Sophus code, resulting in applications that run twice as fast, or more.

1. Introduction

There is a tension between developing efficient programs, and efficient development of programs. In the problem domain of high-performance numerical computation, run-time speed is essential. This has led to a low-level, efficiency-oriented programming style, resulting in programs that are difficult to develop and very hard to maintain. Moving to a more abstract coding style will improve maintainability, but will also have a severe impact on run-time efficiency. Abstract constructs themselves typically generate run-time overhead, but the loss in efficiency is much larger: In our experience (see Section 3), current optimising compiler technology falls short when it sees high-level abstractions. Thus we do not only suffer an overhead by being more abstract, we also lose the optimisations we otherwise would benefit from.

A solution to this problem is to transform the abstract source code into a lower-level, more efficient code. Code-Boost [2, 13, 18] has been developed as a source-to-source transformation tool for C++ [16, 23], intended to bridge the gap between the high-level coding style advocated by modern software engineering, and the lower-level style preferred by current compilers. It is developed as part of the SAGA project, to support the Sophus style of programming [15, 19]. Sophus is a C++ library providing high-level abstractions for implementing partial differential equation solvers.

CodeBoost consists of two components: The transformation framework and the optimiser. The framework has the necessary infrastructure to support transformation of C++ programs, and allows the development of new transformations in the Stratego [26, 27] program transformation lan-

guage or as simpler rules specified in a C++-like syntax [3]. The framework is targeted primarily at supporting the Code-Boost optimiser, which is a high-level, domain-specific optimiser for Sophus.

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.

This paper gives an overview of the design of the Code-Boost framework and the CodeBoost optimiser. The rest of the paper is organised as follows. First, we introduce our underlying design philosophy. Then, we introduce Sophus, and take a brief look at some of the Sophus-specific optimisations that have been implemented with CodeBoost. In Section 4, we discuss the architecture of the Code-Boost framework, followed by a description of user-defined rewrite rules (Section 5) and totem annotations (Section 6). In Section 7, we discuss our experience with transforming C++. Finally, we discuss related work (Section 8) and offer some concluding remarks (Section 9).

2. Design philosophy

CodeBoost is a set of modules written in Stratego to perform improvements on C++ code written in the Sophus style. The dependency on Stratego, Sophus and C++ has naturally formed much of our work on CodeBoost. The development of CodeBoost has been an evolutionary process, adapting to the complexity of C++, the rapid evolution of Stratego and the changing demands of Sophus. Our overall design philosophy is as follows.

Implement only what is needed. C++ is a large language, and writing a complete and fully compliant implementation would take far more time than we had available. It is better to implement only what is needed for the task at hand, and then, later on, extend as required.

Make incremental changes. When changes are made in small steps, it is easier to verify that they work as intended. Also, it is motivating to work on a system which is "alive" and working, instead of programming for long stretches of time with no end in sight. We believe the motivation factor is quite important.

Don't be afraid to make changes. Designing and building a system often requires knowledge which is only available when one has already built such a system. During the course of development, several flaws in the original Code-Boost design have become apparent, and many parts of CodeBoost have been rewritten several times. Some of the changes have been quite intrusive, particularly changes in the internal representation, but still, we have usually been able to complete them in few days.

Modular design. We have three reasons for choosing a modular design. First, by separating code that performs separate tasks into separate modules that communicate through a well-defined interface, we can change or even reimplement one module without affecting the others. For instance, the overloading resolver can be changed without affecting other parts of semantic analysis. Secondly, the order in which optimisations are applied can often be important. Implementing each transformation as a separate module allows us to experiment easily with different orderings. Our third reason is motivated by purely practical concerns: The early versions of the Stratego compiler were quite slow, and used huge amounts of memory when compiling even moderately sized modules. With small modules, we did not have to recompile all of CodeBoost when making small changes.

Testing. Good automated unit and regression tests ensure that a system works as expected, and that undesired side-effects of changes are easily discovered. It has been our goal to provide automated tests for all CodeBoost modules. The currently implemented tests are not as extensive and complete as we would like them to be, but they have helped uncover countless mistakes and bugs.

3. Optimising Sophus

Sophus [15, 19] is a C++ library providing high-level abstractions for implementing partial differential equation (PDE) solvers. This is a field in which runtime performance is of great importance. Unfortunately, although a high-level programming style is beneficial in terms of programmer productivity and program maintainability, it has a large negative impact on performance. Current compilers have proven unable to sufficiently optimise programs in this style, partly due to low demand for such optimisations, and partly because some of the most effective optimisations go beyond the C++ standard. Building a domain-specific optimiser for Sophus allows us to experiment with optimisations, and bridge the gap between the Sophus coding style and current compiler technology. CodeBoost was originally designed to fill this need, and optimisation of the Sophus library remains our primary purpose for developing Code-Boost.

A few of the Sophus design choices relevant for this paper include

- An algebraic programming style with the use of side-effect free expressions and explicit assignments. This is closer to the notation of the mathematical domain. It is also similar to the notation advocated by the functional programming community.

- The use of user-defined array-like data structures with numerical operators on them. This allows us to eliminate the use of explicit loops in the code. So we would write x = a + b; for arrays x, a and b, rather than

stepping through the elements of the arrays in an explicit loop when adding them.

The combined use of these techniques gives problems. The array size may be 0.7MB for small arrays, increasing to 6.4MB for larger examples. When these are manipulated in large C++ expressions, the compiler has to create many large temporary variables to hold intermediate results. Initialising temporaries is expensive, and so is copying return values out of functions. Also, since loops are hidden within separate functions, the compiler may not be able to perform the standard loop optimisation and loop merging tricks it normally does to improve performance.

Further, in conventional numerical programming certain expressions are considered so important that there exist special, highly optimised procedures for them. For example, consider

```
x = mvmult(m,x) + y;
```

for array structures matrix m and vectors x and y, where mvmult is matrix-vector multiplication (and + is vector addition as described above). Here the compiler would generate two temporary variables. This assignment statement can be replaced by the optimised procedure call

```
axby(m,x,y);
```

which performs the calculation and mutates (changes the value of) the variable x to contain the result. Mutating the variable eliminates the need for compiler-generated temporaries.

So, we see that we need the following activities to transform the abstract program back to a low-level optimisable program:

Mutification: Gain control over the generation of temporary variables and copying of return values by replacing algebraic-style expressions by mutating procedures. These rules correspond to built-in C conventions like rewriting an assignment x = x + a; to a mutifying procedure x += a;, but extended to user-defined types.

User-defined rules: Allow very domain-specific optimisations to be defined by the user, e.g., replacing an assignment by axby as sketched above.

Totems: To insert and trace invariants and annotations in the code. Such annotations, together with user-defined rules, may allow CodeBoost to, e.g., eliminate a matrix-vector multiplication if it was known that the matrix was a unit matrix (diagonal matrix with only ones on the diagonal).

That such transformations are useful can be seen in Table 1, where the effect on run-time speed of applying mutification

to the isotropic version of the SeisMod seismic simulation application is tabulated. The results were obtained on an SGI Origin 3800, using version 7.3.1.3m of SGI's MIPSpro compiler with optimisation enabled. Even though the transformation is quite simple, mutification still gives a significant improvement over a state-of-the-art C++ compiler: The mutified version is nearly twice as fast as the plain version, and memory use is reduced to 60%.

Small	*time*	*speedup*	*mem*	*relmem*
plain	239.3s	1.0	111504	100%
mutify	121.5s	2.0	67248	60%
Large	*time*	*speedup*	*mem*	*relmem*
plain	7810s	1.0	305424	100%
mutify	4147s	1.8	183136	59%

Table 1. Results for mutification of SeisMod for large and small data sets. *Speedup* relative to the *plain* version; *mem* is memory use in kilobytes; *relmem* is memory usage relative to the *plain* version.

User-defined rules opens up further possibilities. Seis-Mod makes extensive use of elementwise arithmetic operations on arrays. This has been implemented as a number of overloaded operators, each containing basically the same array-traversing loop. We refactored SeisMod, separating out the common traversal code into a single, generic traversal function, one for each array-like type. This gave us a 5% reduction in code size—with more savings expected as new array-like classes are added—but it also made SeisMod more than three times slower (compare the time for *plain* in Table 1 with *plain* in Table 2). With the help of user-defined rules, we were able to completely eliminate this new overhead [3]; see results for *index-opt* in Table 2. Furthermore, since traversals are now clearly identifiable as calls to a particular function, we are able to employ *map fusion*, which combines several successive traversals into one, further improving performance [2]. This achieves the same effect as loop fusion, which the compiler seems unable to apply.

4. Architecture

The CodeBoost framework consists of a parser, a semantic analyser, a library of transformations, and a pretty-printer. Figure 1 illustrates the typical usage of the components in the framework. The C++ code is first parsed, passed on to semantic analysis, then on to user-defined transformations, which can be applied as many times as necessary, before C++ code is produced by the pretty-printer. The semantic analysis phase may be bypassed if the transformations do not require semantic information.
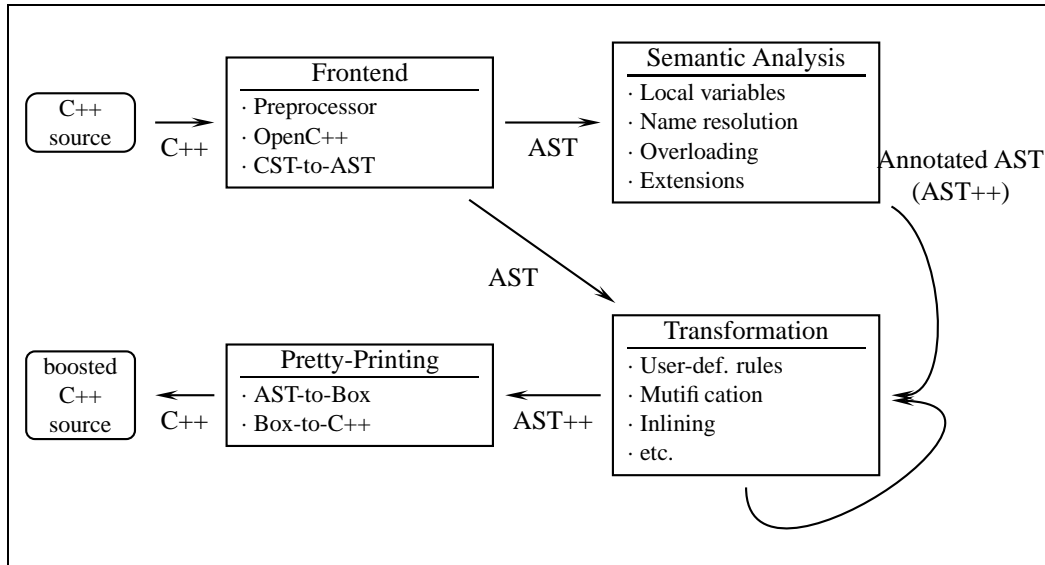
Figure 1. The Transformation Process

Small	time	speedup
plain	827s	1.0
index-opt	111s	7.5
mapfusion	103s	8.1
Large	*time*	*speedup*
plain	25435s	1.0
index-opt	3996s	6.4
mapfusion	3866s	6.6

Table 2. Result for refactored SeisMod with abstracted traversal. *Plain* **is results without CodeBoost optimisation.** *Index-opt* **includes optimisations to eliminate extra overhead in the new-style code;** *mapfusion* **combines multiple successive traversals into one.** *Speedup* **is speedup factor relative to** *plain***.**

4.1. Frontend

The frontend consists of a preprocessor, a parser, and a postprocessor. The C++ preprocessor is often used for *conditional compilation*, in which certain parts of a program are omitted or changed based on options given to the compiler. This is useful when writing programs that should compile on different systems, and to derive different versions of an application from the same program text. This is frequently used in Sophus: Based on options, the preprocessor selects between parallel and sequential code, selects the size of data structures, and picks classes that most closely model the problem domain.

We envisage two ways of using CodeBoost: To transform a full compilation unit in one step, with conditional compilation performed, all header files[1] included, and all macros processed; or to transform a single header file, postponing conditional compilation, macro processing and header file inclusion until compile time. The latter is particularly useful for Sophus, since the same optimised source can be used to derive several different versions of an application.

To support this, CodeBoost has its own preprocessor, which gives the user control over which parts of the program are transformed by CodeBoost. It is possible to delay inclusion of header files, delay macro processing, and read declarations from files which will not be included in the transformed output. CodeBoost uses a standard C++

---

[1]Except the C and C++ Standard Library: Standard library implementations typically contain vendor-specific extensions that may not be fully compatible with the C++ Standard. Besides, CodeBoost itself does not support all valid C++ constructs, and will likely not be able to process the standard library.

preprocessor to remove comments from the input code, and to process inclusions and macros, when desired.

After preprocessing, OpenC++ [9, 10] is used to parse the program text. The result of parsing is a *concrete syntax tree* in ATerm format,[2] which is further processed by the postprocessor into an *abstract syntax tree* (AST). The AST is CodeBoost's preferred internal program representation. In addition to representing the syntactic structure of the program, the AST also contains semantic information resulting from semantic analysis.

4.2. Semantic analysis

The purpose of semantic analysis is to connect symbols to declarations, and annotate each symbol with semantic information. Variables are annotated with their type and the scope in which they are declared. Functions (in function calls) are annotated with the full name and signature of the called function, and type names are qualified fully, so that the type name unambiguously refers to a single type. Annotating function calls can be a bit tricky, since C++ allows overloading; multiple functions can be defined with the same name, as long as their formal argument lists differ. CodeBoost performs overload resolution, and also performs template argument deduction, so it will handle calls to template functions properly.

Since CodeBoost is supposed to be able to handle incomplete programs (e.g., in optimising a single header file) it will not stop when it encounters an undeclared symbol. However, later transformation stages will often rely on semantic information. For example, if the inliner is to inline a particular function call, it needs to know which function definition is associated with that call. Similarly, user-defined rewrite rules (see Section 5) perform matching based on function signatures. If the signatures are incomplete, rewriting will not happen. This is not a huge problem, since it only applies to expressions containing unresolved symbols—calls to the standard library, for instance. However, when processing uninstantiated templates, which is sometimes useful, type information for the template arguments is unavailable. This can cause CodeBoost to miss some optimisations due to lack of information.

4.3. Transformations

In CodeBoost, transformations are implemented as modules. Each module is a separate program, which reads, transforms and writes an AST. The modules are connected

together with Unix pipes to form a transformation pipeline. Examples of transformation modules include the mutification optimisation from Section 3, and the inliner, which inlines particular function calls in support of other optimisations.

The CodeBoost framework contains a transformation library to ease the implementation of transformation modules. There are strategies for symbol table lookups, type comparison, matching and conversion, various kinds of traversal, and simple pretty-printing for error messages, warnings and debug information. Additionally, Stratego has a comprehensive library of generic, language-independent strategies which are useful when implementing CodeBoost modules.

The separation of traversals and transformations in Stratego makes it possible to extract parts of a larger transformation and generalise them. Such transformations can then be added to the general CodeBoost library and reused for other purposes.

User-defined rewrite rules are specified within the C++ program text, and can be written without any knowledge of Stratego. CodeBoost will pick up these rules, and make them available to transformation modules written in Stratego. Two modules included in CodeBoost, `simplify` and `apply-user-rules`, allow for application of user-defined rules. Such rules can be used to exploit fundamental equalities in a program, by specifying program- or library-specific optimisations.

4.4. Backend

After transformation is complete, the pretty-printer converts the AST back into C++ program text. The output program is properly indented, and should be fairly readable to humans. Alternatively, the result of transformation can be stored as an AST, and be loaded into CodeBoost later on. This can be advantageous in a setting where a C++ header file is used by several other C++ files; instead of analysing and transforming the same header file several times, it can be processed just once. Loaded ASTs will not be subjected to parsing and semantic analysis, and later transformation stages will typically not re-transform such previously processed code (though the code may be used by e.g., the inliner).

5. User-defined rules

There are two ways to add new transformations to CodeBoost: writing a new Stratego module, or specifying user-defined rules. We will not go into detail on specifying transformations in Stratego, as this is discussed at length elsewhere [2, 8, 17, 20, 25]. Stratego is useful for implementing complex, generic transformations, whereas user-

---

[2] ATerm (Annotated Term Format) is a format for exchanging structured data between tools [7]. Stratego uses ATerms as its term representation. ATerms are supported in the C, Java and Haskell languages through the ATerm Library. However, the term format is easy to parse, so an ATerm reader/writer can be written for other languages as well.

defined rules are appropriate for simpler domain-specific transformations. A typical application of user-defined rules is domain-specific optimisations where certain combinations of function calls can be simplified or replaced with calls to special-case optimised functions.

Domain-specific optimisations often rely on rewriting calls to specific functions. This is problematic when specifying rewrite rules. In a language such as C++, which allows function overloading, it is not sufficient to simply specify the name of a function; to uniquely identify a function, its full signature—including argument types—must be specified. This is tedious and error-prone, particularly when working with the abstract syntax in Stratego.

In CodeBoost, we have solved this by allowing rules to be specified in C++ syntax, within the C++ program text. Function calls in rule patterns are subjected to normal overload resolution; thus, the correct signature is automatically deduced.

A user-defined rule consists of a name, a match pattern, a replacement pattern and an optional condition. Rules are contained within a function named `rules()`; such functions can be placed anywhere in the program, and are removed by CodeBoost after the rules have been extracted. A typical rule looks like this:

```
int x;
simplify: x + 0 = x;
```

where 'simplify' is the name, 'x + 0' is the match pattern and 'x' is the replacement pattern. Local variables are treated as meta-variables, and will match anything. Additionally, list matching is available for use with functions that accept a variable number of arguments.

In the above example, the + will resolve to the integer addition operator, and the rule will match any integer addition of the form $x + 0$, and rewrite to x. For example, when the rule is applied to (3 + 2) + 0, x will match (3 + 2), and the result will be (3 + 2).

Conditions are separated from the rest of the rule with a comma. The condition must succeed for the rule to be applied. For example, a more advanced version of the previous example would be

```
int x, y;
simplify: x + y = x, is_zero(y);
```

where `is_zero` would use some kind of data flow information, such as totems (discussed in Section 6) to determine if y has a zero value. A number of built-in condition checks are available; it is also possible to call other user-defined rules from conditions. In keeping with our design philosophy, we have only implemented the few built-in conditions we have needed for optimisations; others can easily be added with a few lines of Stratego code.

Rules with the same name are collected into *rule sets*. Rule sets are callable from Stratego programs, and from rule conditions. When a rule set is applied, each of its rules is tried, until one is successfully applied. If no rule applies successfully, the rule set fails.

It is sometimes useful to write rules which are applied only to see if they succeed or fail. For example, the following two rules sort addition expressions so that multiplication sub-expressions are on the left side:

```
sort: z + (x * y) = (x * y) + z,
             not(is_mult_expr(z));
is_mult_expr: (a * b) = true;
```

In the rule `sort`, `is_mult_expr` is used to check if the left-hand side already contains a multiplication expressions; if so, the rule fails. This is particularly useful in *generic rules*, in which functions (and their signatures) are matched with meta-variables, thus matching any function, not just a particular function. For example, given a rule set `is_commutative` that applies successfully to commutative functions, we can write a generic rule that employs commutativity laws, and uses `is_commutative` in a condition to check for commutativity.

User-defined rules in CodeBoost is discussed in more detail in [3].

6. Totem annotations

For domain-specific optimisation to work, the optimiser must have access to domain knowledge. This knowledge can be in the form of executable transformations, written in Stratego or as user-defined rules. Another option is to use declared domain knowledge, where the programmer gives hints to the optimiser by declaring properties of variables and functions. We have implemented a mechanism in Code-Boost for tagging language constructs with auxiliary information, and propagating these tags throughout the program. This can be used, for example, to specify that a matrix is a diagonal matrix; combined with a simple user-defined rule, this gives us a transformation that automatically selects efficient, specialised versions of matrix operations.

In our nomenclature, such a tag is called a *totem*, as the tag is usually an emblem signifying membership in a special class of entities (e.g., diagonal matrices).

CodeBoost recognises function calls to `CB_TAG` and `CB_IMPORT` as special totem directives. A totem is typically either a switch or some kind of meta information used by transformations.

Arbitrarily complex meta information stored in a configuration file separate from the source code can be loaded at boosting time by the `CB_IMPORT` directive. Each external configuration file may contain many named bundles of meta information, each called a *context*.

After the contexts are loaded, they may be tagged onto entities by their name, using the `CB_TAG` directive. Subsequent transformations are then free to inspect the various entities for the presence of relevant meta information.

The content of the meta information is opaque to Code-Boost; its semantics are known only by the transformations. An example of useful meta information is layout information for matrices, which can be used in partial evaluation of matrix operations [18].

In some cases, it is reasonable for totems to propagate throughout the program. In the code fragment

```
Matrix A, B;
CB_TAG(A, "simplification",
        "unit-matrix");
B = A;
```

the matrix `A` is tagged with the *unit-matrix* totem in the *simplification* context. We know from mathematics that matrix `B` should inherit the *unit-matrix* property from `A`. For this, we have implemented some rudimentary strategies for generic totem propagation that;

- Propagate totems across the assignment operator.

- Drop totems on variables that are potentially modified (i.e., the variables are assigned to, or passed as non-`const` parameters to functions).

After totem propagation, we can apply simplifying transformations that are aware of the properties available in the *simplification* context (which additionally could contain flags for diagonal, triangular, checkerboarded or other common matrix layouts). They may issue queries to the framework for the unit matrix property on matrix variables, and can thus simplify the code accordingly.

This simple propagation approach will only work for a very limited set of totems. For the more involved totems, separate totem-propagation strategies that accompany a particular transformation need to be written.

7. Discussion

7.1. Transforming C++

C++ is a large and complex programming language. It supports many different programming paradigms and styles, including low-level C-like programming, the algebraic style advocated by Sophus, object-orientation, and generic programming. This flexibility gives a lot of freedom to programmers, but makes the implementation of language processing tools such as compilers and transformation systems extra challenging. The main problems with C++ are (from an implementor's perspective):

- The grammar is large, and not context-free. This makes it difficult to implement a correct parser using standard tools.

- C++ has a lot of features. This is a problem both because there are more features to implement, and because language features have a tendency to interact in interesting ways.

- The programmer is free to mix high-level and low-level coding styles. High-level optimisations will often not work on low-level code, and low-level features—such as pointer manipulation—can confuse or complicate the analysis needed to support high-level optimisations.

CodeBoost provides no satisfactory solution to the first problem. The OpenC++ parser does not correctly resolve the ambiguities in the C++ grammar, and will give incorrect results in certain cases. This has not been a huge problem so far, because the ambiguous constructs are seldom used in Sophus. We briefly considered developing our own parser based on the syntax definition formalism SDF [24], but it would have taken too much time. Recent efforts, however, may provide us with a good (and correct) SDF-based parser [1].

We have dealt with the second problem by limiting ourselves to a manageable subset of C++. Language constructs that are used by Sophus, but are irrelevant to optimisation, can be left untouched, and some features that are not needed, can be ignored.

In developing high-level optimisations for Sophus, we have solved the last problem by placing additional semantic restrictions on the language. For instance, the C++ standard specifies no special semantic relationship between + and +=, or between construction and assignment. Sophus, on the other hand, defines such relationships, and this is part of what allows CodeBoost to perform effective, high-level optimisations.

7.2. Maintainability and optimisation

In the process of developing software, programmers make tacit assumptions about the code. Such assumptions include invariant properties, such as *this should now be a diagonal matrix*, or properties about the relationship between functions/methods, such as *these two functions are inverses of each other*. By recording these assumptions in the program text, maintenance is eased, as the intent of the code becomes clearer. Such annotations may also be used to verify that the properties assumed actually hold, e.g., testing whether an assumed diagonal matrix actually is a diagonal matrix. But the annotations may also be given a dual reading, as information which a program transformation system

may use for optimising code. We have primarily focused on the latter aspect.

In our experience, the use of CodeBoost (and high-level optimisation in general) encourages the writing of shorter programs and more maintainable code. Without sufficient automated optimisation, it is often necessary to hand-inline and hand-tune code to get acceptable performance. This in turn makes the code more difficult to understand and to modify. With CodeBoost, we can write high-level, easy-to-understand code, and still get good performance, by the use of domain-specific, user-defined optimisations.

As for maintenance of the optimisations themselves, user-defined rules and totems are typically kept close to relevant parts of code, and require little extra maintenance work. Since rules are often used to capture fundamental relationships between functions, many rules will be correct independently of the concrete implementations of functions. As noted above, such rules may serve a dual purpose as both assertions of expected behaviour, and as optimisation rules.

7.3. Correctness of transformations

The programmer is responsible for keeping rules and totems correct and up-to-date with respect to the rest of the program. CodeBoost makes no attempt to verify their correctness—after all, the point of such annotations is to allow the programmer to communicate knowledge that the optimiser is unable to deduce on its own. Rules will typically follow from the specification of a module or design of an algorithm, and will need to undergo normal program validation techniques—along with the rest of the program—to ensure validity.

8. Related work

For numerical software the TAMPR program transformation system [5, 6] has been used with remarkable success. Its main use has been the specialisation of numerical library code from generic code, but it has also been used for optimisation of code.

The algebraic specification formalism ASF+SDF [12] has strong syntactic capabilities, and supports rewriting with high-level transformation rules. It was used in a first experimental version of CodeBoost [13]. One of the observations in this project was that pure rewriting on constructors of abstract trees was not sufficient for the context-sensitive application of transformation rules required in program optimisation. Spelling out the traversals over the complex syntax trees of C++ programs would be needed to get control over the application of transformation rules, but proved cumbersome to specify. For the same reason, specifying the complex semantics of C++ would be difficult. It is

this aspect where the generic traversals of Stratego provide an advantage.

Several transformation systems for C++ exists. OpenC++ [9, 10] provides a meta-object protocol for C++. A meta-object protocol is an object-oriented interface for specifying language extensions and transformations. As such, OpenC++ has many of the same capabilities as CodeBoost, and can be used to implement domain-specific optimisations, as well as other transformations, in an object-oriented fashion. However, it does not support cascading transformations (i.e. applying other transformations to the output of previous transformations), and specifying transformations in C++ is cumbersome compared to using a domain-specific transformation language such as Stratego.

Sage++ [4] is a C++ toolkit for source-to-source transformation of C++ and Fortran. It provides classes for representing the various nodes of a program tree, and tools for data flow analysis and loop transformations. However, it offers none of the support for easy rewriting and traversal one would expect from a transformation language; matching and building of terms must be hand-coded in C++.

ROSE II [11] is a transformation tool for optimising object-oriented C++ code. Its primary focus has been on developing optimisations for array classes. Targeted optimisations include loop fusion, cache-based optimisations, temporal locality optimisations and the introduction of performance-gathering options and metrics.

Simplicissimus [21, 22] shares many of the same goals as CodeBoost. It is implemented as a plug-in to a compiler (currently GCC, but other compilers may be supported as well), and supports user-specified rewriting of expressions. The compiler's template mechanism is used for pattern matching, to aid rewriting. Simplicissimus allows some degree of strategic control over the application of rules, through its *arbiters* (deciding which rules are applied), *stages* (deciding when they are applied), and *directors* (deciding how the program is traversed). The functionality of Simplicissimus is similar to that of user-defined rules in CodeBoost.

The Broadway compiler [14] allows library designers to annotate their libraries with semantic information that will be used in high-level optimisations. The compiler is focused on the numerical domain, where for instance a high-level program may require the solution of a linear system of equations. There exist many variations of such equation solvers, and the more that is known about the properties of the linear system, the more efficient the solver algorithm. The Broadway compiler tries to automatically select optimal solvers by using annotations from the solver library to track properties of the data in the high-level program. The Broadway compiler uses principles similar to our totem functionality, but is more advanced.

9. Conclusion and future work

In this paper, we have given a brief overview of the Code-Boost source-to-source transformation system. The Code-Boost framework provides the basic infrastructure needed to apply transformations to C++ programs. CodeBoost is extensible and can be used for experimentation with optimisations, as well as other kinds of transformations. It supports cascading transformations; any number of transformations can be applied in any order, in one or multiple passes over the syntax tree, without having to pretty-print and reparse. CodeBoost also allows C++ programmers to specify domain- and program-specific transformations through user-defined rewrite rules.

To support high-level C++ optimisations, we have basically had to build a limited C++ compiler frontend. For a language as complex as C++, this is a large undertaking. We believe that our design philosophy has helped ensure our success, particularly in reminding us to keep things simple. For example, aiming at full C++ standards compliance would be infeasible given our limited resources; therefore, we have limited ourselves to the parts that are strictly necessary for Sophus. This does not mean that CodeBoost is useful only for toy examples; the Sophus application Seis-Mod, which we have successfully optimised using Code-Boost, consists of over 14000 lines of code.

The primary purpose of CodeBoost is to bridge the gap between the high-level Sophus style, and the coding style expected by optimising compilers, thus allowing the use of high-level abstractions with little or no performance penalty. With the optimisations we have implemented so far, we have achieved significant speedups, beyond what normal optimising compilers achieve.

Our most promising development is the user-defined rules, which will allow easy implementation of future optimisations, especially combined with the totem annotation mechanism. Future development efforts will focus on extending the rule language, making it suitable for the specification of advanced transformation. Additionally, we intend to evolve the supporting framework, refactoring as needed, and moving in the direction of standards compliance. Better C++ support will make CodeBoost more useful for non-Sophus projects.

CodeBoost is Free Software, and can be freely modified and extended under the GNU General Public License. For more information, see the CodeBoost web page: `http://www.codeboost.org/`.

10. Acknowledgements

References

[1] R. Anisko. Program transformation and the C++ language. Technical Report 0217, LRDE, EPITA, France, 2002.

[2] O. S. Bagge. CodeBoost: A framework for transforming C++ programs. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[3] O. S. Bagge and M. Haveraaen. Domain-specific optimisation with user-defined rules in CodeBoost. In J.-L. Giavitto and P.-E. Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03)*, volume 86/2 of *Electronic Notes in Theoretical Computer Science*, Valencia, Spain, June 2003. Elsevier Science Publishers.

[4] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings of the Second Annual Object-Oriented Numerics Conference (OONSKI'94)*, 1994.

[5] J. M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.

[6] J. M. Boyle, T. Harmer, and V. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, Boston, 1997.

[7] M. G. J. v. d. Brand, H. d. Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[8] M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, Lecture Notes in Computer Science, Copenhagen, Denmark, July 2002. Springer-Verlag.

[9] S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299. ACM, Oct. 1995.

[10] S. Chiba. Open C++ programmer's guide for version 2. Technical Report SPL-96-024, Xerox PARC, 1996.

[11] K. Davis and D. Quinlan. ROSE II: An optimizing code transformer for C++ object-oriented array class libraries. In *Third World Multiconference on Systemics, Cybernetics and Informatics (SCI'99) and the Fifth International Conference on Information Systems Analysis and Synthesis (ISAS'99)*, Orlando, FL, USA, 1999.

[12] A. v. Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[13] T. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*, 8(4):247–259, 2000.

[14] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of DSL'99: The Second Conference on Domain-Specific Languages*, Austin, Texas, USA, 1999. The USENIX Association.

[15] M. Haveraaen, H. A. Friis, and T. A. Johansen. Formal software engineering for computational modelling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

[16] ISO/IEC JTC1 SC 22. *ISO/IEC 14882: Programming languages — C++*, 1998.

[17] P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.

[18] K. T. Kalleberg. User-configurable, high-level transformations with CodeBoost. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[19] H. Munthe-Kaas and M. Haveraaen. Coordinate free numerics — closing the gap between 'pure' and 'applied' mathematics? *Zeitschrift für Angewandte Mathematik und Mechanik*, 76, supplement 1:487–488, 1996.

[20] K. Olmos and E. Visser. Strategies for source-to-source constant progagation. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier Science Publishers, 2002.

[21] S. Schupp, D. P. Gregor, D. R. Musser, and S.-M. Liu. User-extensible simplification—type-based optimizer generators. In R. Wilhelm, editor, *International Conference on Compiler Construction*, Lecture Notes in Computer Science, 2001.

[22] S. Schupp, D. P. Gregor, D. R. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, October 2002.

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

[24] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[25] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[26] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[27] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.