CodeBoost

A Framework for Transforming C++ Programs

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

Abstract. Often we are faced with the need to make trivial, albeit tedious, changes to program code. It may be things like making variable names more readable, add code that will provide execution profile information, or change the style of a program from from expression oriented to object oriented in order to improve run-time efficiency. Such source-to-source transformations can be aided by, or even completely automatised, with the aid of a suitable program transformation tool. Here we present the CodeBoost framework for the implementation of source-to-source transformation of C++ programs. It is implemented using OpenC++ for the syntax analysis and using Stratego for defining the program transformations. Stratego allows for the easy expression of context sensitive transformations, a central point when using transformations to improve execution speeds of code. We also discuss two example applications.

1 Introduction

Source-to-source transformations can be used in various ways to increase programmer productivity. Source-to-source optimisation can be used to reduce the overhead caused by an abstract, high-level style of programming, or to achieve domain-specific optimisations using domain knowledge that the compiler does not have access to [9]. Instrumentation can be used to systematically extend all functions of a program to obtain profiling or debugging versions [2]. Refactoring transforms a program to improve its design [10]. Specialisation transforms a generic piece of code to a situation dependent program, e.g., instantiating type declarations to taking into account knowledge of input data [4]. Also, (domain-specific) language extensions can be implemented by means of transformations to the core language supported by the compiler.

The implementation of such source-to-source transformations requires a considerable language processing infrastructure capable of parsing and pretty-printing programs, performing semantic analysis and implementing the transformations themselves. In particular for a complex language such as C++ [12] this requires a large effort.

In this paper we describe CodeBoost, a framework for the implementation of source-to-source transformations of C++ programs. CodeBoost was originally motivated by the need of the Sophus numerical software library [11] to optimise effects of using a very high-level style of programming when writing high performance numerical code [9].

When doing source code optimisations, the transformations are very sensitive to the context where they may be applied. The Stratego language for program transformation [16] is well suited for this. In Stratego the description of the transformations, which may become quite complex, are separated from the description of the context in which they may be applied. Transformations are described by a combination of concise rewrite rules and strategies specifying how and where the rules are applied. The rules are independent of the overall transformation, and can be reused for other transformations. Strategies can be parameterised by transformations to apply, making it possible to develop libraries of reusable transformations.

CodeBoost performs semantic analysis, and supports function and operator overloading and templates. CodeBoost is extensible, supports cascading transformations, and can be used for experimentation with optimisations, as well as other kinds of transformations.

The paper is organised as follows: Section 2 describes the architecture of the framework. In section 3 we discuss the implementation of two example applications with the framework: instrumentation and domain-specific optimisation. In section 4 we present the specification of an optimising transformation.

2 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 proper C++ code is produced by the pretty-printer. The semantic analysis phase may be bypassed if the transformations do not require semantic information.

2.1 Parsing

The CodeBoost front-end converts C++ code to the abstract syntax tree (AST) format used internally. A combination of Perl and the preprocessor of a C++ compiler is used to preprocess the code. The preprocessor allows the user to specify which parts of the code will be touched by CodeBoost. It is possible to preserve sections of the code verbatim, or delay inclusion of header files.

The parsing is done by the parser of OpenC++. The parser has been modified to output ATerms [13]. The ATerm library supports C and Java, but a parser can easily be written for most languages. Because OpenC++ works on a concrete syntax tree, a second pass is necessary to convert to AST format. The AST can

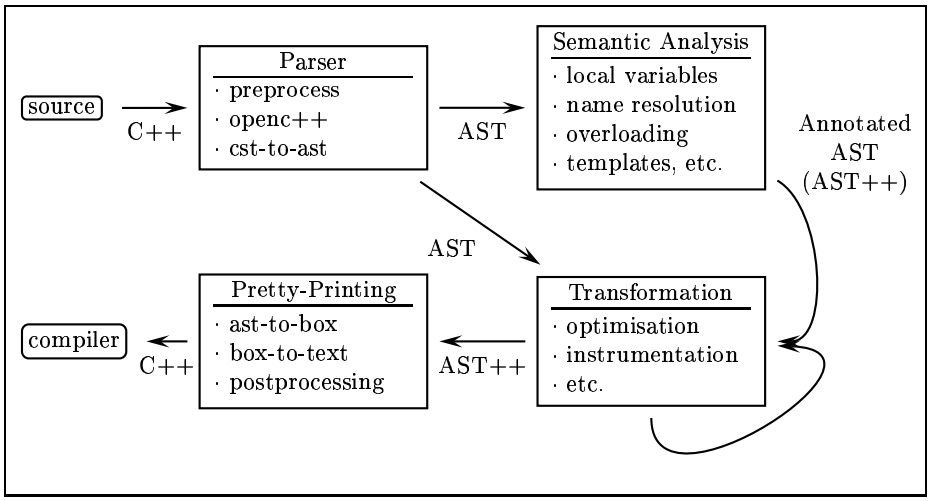

Fig. 1. The Transformation Process

then be fed directly to a user-defined transformation component, or semantic analysis can be performed, if that is required. A sample AST fragment is shown in Figure 2.

2.2 Pretty-Printing

The pretty-printer converts the AST back to C++ code. The AST is first transformed to Box, a device and language independent format for pretty-printing [14]. We may use the GPP system pretty-printer [7], or a simpler ABox formatter. The output is properly indented, and reasonably readable for humans.

In addition to plain text output, CodeBoost also provides facilities for highlighting parts of the code, and producing PostScript (with GNU `a2ps`). This is useful for visualising properties of the code, such as execution profiling results.

```
if(a)
    ++b;
```

↔

```
If(Var(IdName("a"),AnyType,NoScope),
   Expr(Prefix(Op("++",AnyFun),
               Var(IdName("b"),AnyType,NoScope))))
```

(a) C++ (b) AST

Fig. 2. Example of C++ code, and the corresponding AST

2.3 Semantic Analysis

The semantic analyser consists of three parts. The first part uses a simple environment-passing scheme to annotate local variables with their types. The second part collects all non-local declarations into a symbol table and resolves variable names. Function and operator applications are annotated with a list of possible candidates. The last part performs overload resolution, computing the type of all expressions and, for each application, selecting the correct function from the list of candidates.

It is not always desirable to transform an entire program, especially since CodeBoost does not support all C++ features (like namespaces and inheritance). Typically, the standard library will be unsuitable for transformation, or contain vendor-specific extensions that would disappear during parsing. Inclusion of standard library header files will therefore often be postponed until compilation time. CodeBoost has declarations for C++ built-in operators, and some of the standard library functions, and will fall back on them if no other declarations are found. Undeclared variables and functions are accepted, but missing type information may cause trouble for later transformations.

After semantic analysis, variables will have type information attached (instead of `AnyType` as the type information available for the variables `a` and `b` in Figure 2), and function calls have been annotated with a copy of the declaration of the function (replacing the generic `AnyFun` in Figure 2).

2.4 Transformation

User defined transformation plug-ins are separate components, connected to the framework by pipes or intermediate files. The components are typically specified in Stratego, which works directly on ATerms, but it is possible to use other programming languages to specify transformations.

The CodeBoost framework contains a library of Stratego modules to ease the implementation of C++ transformations. There are strategies for symbol table lookups, type comparison, matching and conversion, various kinds of traversal, and simple pretty-printing for error messages.

Stratego has a library of generic, language independent strategies, including renaming of bound variables [15]. An instantiation of this for C++ is available in CodeBoost. Optimisations such as inlining and loop fusion are implemented as part of the Sophus optimiser project, but can be reused for other projects as well.

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.

3 Example Applications

3.1 Instrumentation: Path profiling

Instrumentation of a program is useful for investigating correctness, debugging and learning about its run-time behaviour. A program may be instrumented to check that data invariants hold at critical places in the code, or to produce a stack trace if it crashes. Code can be added to a program for coverage analysis: showing which parts of a program have been executed, so that testing code can be improved. Performance profiling can identify bottlenecks, and give hints on where to concentrate optimisation efforts.

Path profiling counts the number of times each of the possible execution paths in a function is executed. This can be used for path coverage analysis, where we check whether all paths are executed, or for program optimisations, where we are interested in which paths are executed the most.

The code segment shown in Figure 3 has three possible paths, which are enumerated 3 (`a` is true), 2 (`a` is false and `b` is true), and 0 (`a` and `b` are false). On the right of the figure, path 3 has been shaded. In Figure 4 code has been inserted to count the traversals of each path. The function `P(p,`n`)` sets bit n of `p`, so at the end of the segment the integer variable `p` will identify which path was executed. The tally of this path is then increased in the last statement.

An instrumentation tool for path profiling has to analyse the possible paths through the code, identify each path, and then insert code which will tally the execution of a path. In our implementation, a simple bit-numbering scheme is used to number the paths: A single bit is assigned to each control-flow construct

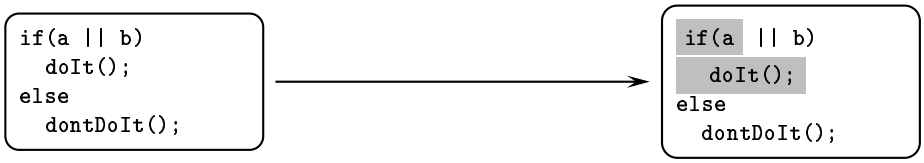

Fig. 3. Path Profiling. The right box shows how a single path or subpath can be illustrated by shading the source code: The first part of the `||`-expression, and the *then*-part of the `if` are in path 3.

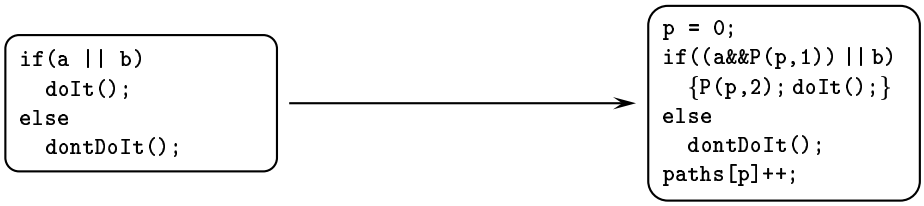

Fig. 4. Instrumentation for Path Profiling. The right box shows how the code segment on the left has been modified to count how many times each path is executed.

(`if`, `&&`, ...) in a function. Together, the bits form a number uniquely identifying a path through the function. This technique is simple to implement, but wastes path numbers. Ball and Larus[2] have developed a more efficient scheme, which also tries to place the instrumentation in the least executed paths.

Path profiling requires only syntactical information, so semantic analysis can be skipped, see Figure 5. Instead we need to provide two kinds of output: the instrumented code which goes to the compiler, and a visualised version where each path through the code segment can be identified (the enhanced C++ text that goes to the user). In our implementation we only shade one path at a time, so the path number to be visualised is provided as an extra parameter to the transformation unit. The CodeBoost path profiler is then run in one of two modes: either as an instrumentation tool which adds the profiling code shown in Figure 4, or as a visualisation tool which, depending on the path number given, will provide a pretty-printed version of the code with one path shaded as shown in Figure 3

The Stratego implementation of this consists of three parts: a generic traversal strategy, control-flow rules, and instrumentation rules. The generic traversal strategy does a top-down traversal of the program, carrying along the current bit number. The traversal order is overridden by the control-flow rules. For each node in the AST, the control-flow rules and then the instrumentation rules are tried. If neither succeeds, traversal continues normally. The control-flow rules specify the propagation of the bit number. For instance, the rule for an `if` statement states that the incoming bit number, b_{in}, should be used for instrumenting this statement. Each branch of the `if` should be traversed with $b_{\text{then}_{in}} = b_{\text{else}_{in}} = b_{in} + 1$. The outgoing bit number, to be used in the next

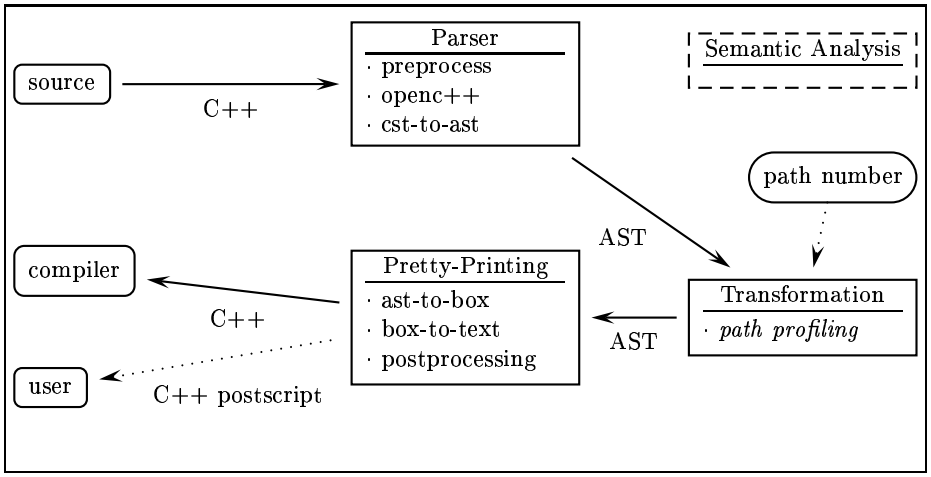

Fig. 5. The Instrumentation Process. Semantic Analysis is bypassed, and the same module is used for both instrumentation and presentation. The PostScript output features of the postprocessor are used for visualisation.

statement, is $b_{out} = \max(b_{\text{then}_{out}}, b_{\text{else}_{out}})$. The rules for other statements are similar.

The instrumentation rules differ depending on whether we are instrumenting a program, or visualising a path. In the first case, for an `if` statement, instrumentation is added to the *then*-branch, setting bit b_{in} of the path number in the function to 1. Together with other rules, this results in code shown in Figure 4. In the second case, we test bit b_{in} of the path number under consideration, and, if set, a `Mark` node is added to the AST, giving output as in Figure 3.

The actual analysis of the path executing profile will be done on the data gathered in the `path` array of Figure 4. Such analysis is not part of CodeBoost.

3.2 Optimisation: The Sophus project

The Sophus project [11] explores the use of high-level abstractions for numerical applications. This has great advantages in terms of programmer productivity and program maintainability, but poor performance has hindered the adoption of these techniques in high performance computing. 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, such as extending the semantic relationship between `+` and `+=` from the built-in arithmetic types to all user-defined types and classes. 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.

An experimental CodeBoost version [9], implemented in the abstract specification formalism ASF+SDF [8], showed promising results, but without semantic analysis, more advanced optimisations were difficult. The new CodeBoost framework is a result of the effort to create a more advanced optimiser.

Sophus uses a coding style that is close to mathematical notation. Programs are written in terms of non-updating functions; any updating is done with the assignment operator. Functions that modify their arguments, *updating (or mu-*

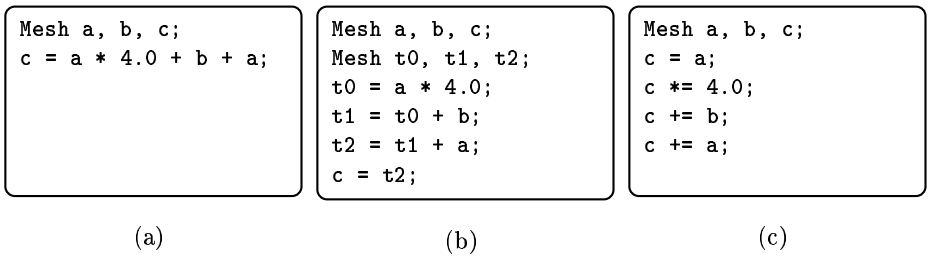

Fig. 6. The code in (a) would normally be evaluated as in (b), with three temporary meshes. Mutification gives code as in (c), eliminating temporaries and copying.

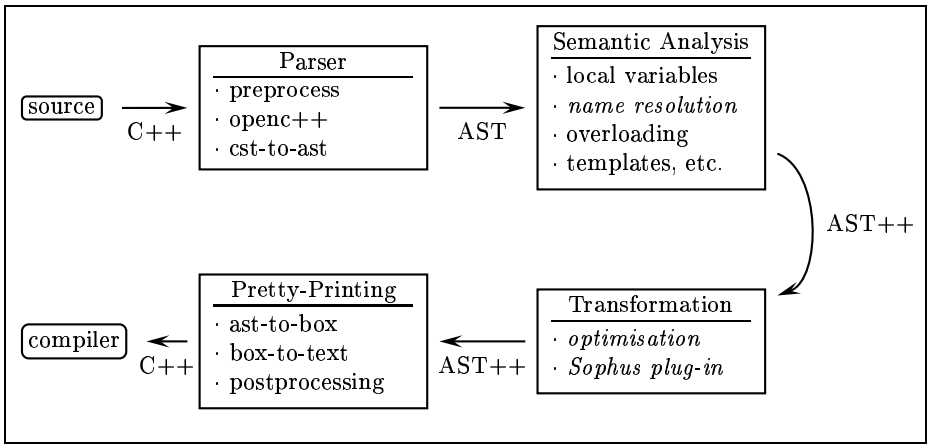

Fig. 7. The Optimisation Process

tating) functions, are forbidden in user code. For our purposes, operators are considered functions.

This mathematical style causes certain inefficiencies in the generated code. Since no Sophus function can update its arguments, the result is returned by value, requiring an extra temporary. As Sophus uses large data structures (several megabytes), the allocation, initialisation and copying of temporaries can have a significant impact on performance. In addition, the heavy use of function abstractions adds function call overhead. *Mutification* (see Figure 6) solves this problem by converting Sophus-style expressions to expressions using only updating functions; the return value is written into an argument. In the Sophus code, only the updating version of a function needs to be implemented, CodeBoost will supply a declaration for the non-updating version.

C++ provides no way to specify the relation between user-defined updating and non-updating functions, so the compiler can only do this for the built-in operators. However, the domain-specific nature of Sophus allows CodeBoost to safely apply this transformation to user-defined functions. Figure 7 shows how this optimisation fits into the framework. The actual implementation of mutification is described in Section 4.

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. An inline component for CodeBoost gives us fine-grained control over the inlining process, and can be useful outside Sophus as well.

Many Sophus library functions iterate over large data structures. In a complex expression, there will typically be many loops, each inside its own function. Doing the computation this way, in several passes, leads to a lot of cache misses;

if we reduce this to a single pass, we can make better use of the cache. This can be done by inlining each function, and then performing *loop fusion* — joining a series of similar loops into one. Loop fusion requires, in addition to syntactic and semantic analysis, data flow analysis to determine if code can be moved without changing the meaning of a program. Loop fusion is already done by certain existing compilers, but they work at a low level, and do not seem able to handle the interplay with inlining functional abstractions.

Further optimisations may also be feasible, such as applying algebraic laws to computations, or taking advantage of known properties of objects (such as matrix symmetry).

4 Specification of Mutification

In this section we illustrate the specification of a transformations in CodeBoost by means of the specification of mutification.

The implementation of mutification consists of two parts. The first part is a modification of the name resolution module: For each occurrence of an updating function, a declaration for the corresponding non-updating 'shadow' function is added to the symbol table. Furthermore, a table is kept, mapping shadow functions back to real functions. No code is ever generated for the shadow functions, so every occurrence must be eliminated before compilation.

The second part is a new transformation module, which is described below. The code presented here is a special case that only deals with operators. The code for the general case is similar, but more complex, because functions can have any number of arguments.

Mutification of operators is defined by the following rules:

$$\texttt{x = x op e;} \rightarrow \texttt{x op= e;} \tag{1}$$

$$\texttt{x = e1 op e2;} \rightarrow \texttt{x = e1; x op= e2;} \tag{2}$$

$$\texttt{x op1= e1 op2 e2;} \rightarrow \texttt{t = e1 op2 e2; x op1= t;} \tag{3}$$

An example of the application of these rules to a C++ code fragment is shown in Figure 8.

The Stratego specification of the mutification rules is shown in Figure 9. A Stratego specification consists of a signature specifying sorts and constructors for the terms to which it will be applied, rules that can be applied to the terms and strategies that specify how the rules are to be applied. The signature for the AST used to represent C++ is specified in a separate module which is imported into all CodeBoost modules by means of the `imports` directive. The other modules imported into the mutification module are the standard Stratego library and the CodeBoost library.

In this example, a special constructor `C(ds,es)` has been added to the AST signature which is used internally to store compound expressions. The first argument is a list of declarations for temporaries, the second argument is a list of expressions. After mutification is complete, the compound expressions will be

transformed to lists of statements or blocks. The other constructors used are `Expr(e)`, which represents an expression as a statement; `Infix(op, r, l)`, an infix operator `op` with arguments `r` and `l`; and `Op(o, d)`, the operator `s`, with the declaration `d`.

Strategies are used to direct the application of rules. The mutification strategy `mutate` specifies that a bottom-up traversal should be used and at every node, one of the `mut0–3` rules should be applied. If one of them succeeds, the result is recursively mutified. The operators `<+` and `;` are used to compose strategies: `s1 <+ s2` means try `s1`, and if that fails, do `s2`. `s1 ; s2` is sequential composition.

Rules can have conditions; a `where` clause with a strategy that must succeed before the rule is applied. Conditions are used here for finding the updating version of a function (`find-mutating-op`) and for generating names and declarations for new temporaries (`make-temporary`).

The rules `mut1` to `mut3` implement rule (1) to (3) above. In rule `mut1` we match against the pattern `x = x o e`. If a match is found, `find-mutating-op` is applied to `o` in order to find an updating version of the operator. The term is then rewritten as `x o= e`.

The second rule, `mut2a`, is similar: we match against `x = e1 o e2`. An extra test is done to ensure that `x` is not used in `e2`. We use the `C` constructor to make a compound assignment, which will later be flattened, to produce plain, linear C++ code. In Sophus, the assignment operator (`=`), is assumed to have its customary meaning, so we can ignore its declaration when we build a new assignment.

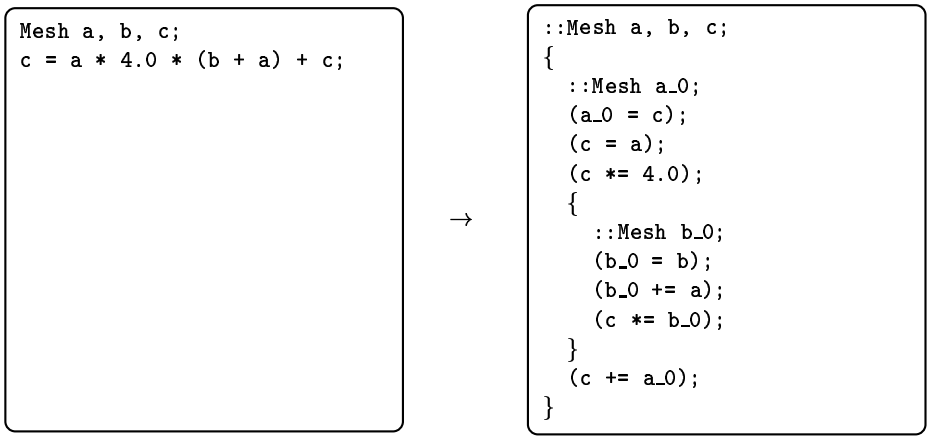

Fig. 8. The code in (a) is shown after mutification in (b).

```
module mutification
imports lib cpp symtable names types cb sigs shadows
signature
  constructors
    C : List(SD) * List(SD) -> SD

strategies

  mutate = bottomup(try((mut1 <+ mut2a <+ mut2b <+ mut3); mutate))

rules
  mut1:
    Expr(Infix(Op("=", _), x, Infix(o, x, e))) ->
    Expr(Infix(Op(o', d), x, e))
    where <find-mutating-op> o => Op(o', d)

  mut2a:
    Expr(Infix(Op("=", _), x, Infix(o, e1, e2))) ->
    C([], [Expr(Infix(Op("=", AnyFun), x, e1)),
           Expr(Infix(Op(o', d), x, e2))])
    where <not(used-in)> (x, e2);
          <find-mutating-op> o => Op(o', d)

  mut2b:
    Expr(Infix(Op("=", _), x, Infix(o, e1, e2))) ->
    C([tdecl], [Expr(Infix(Op("=", AnyFun), t, x)),
                Expr(Infix(Op("=", AnyFun), x, e1)),
                Expr(Infix(Op(o', d), x, e2'))])
    where <used-in> (x, e2);
          <make-temporary> x => (t, tdecl);
          <replace-var> (x, t, e2) => e2';
          <find-mutating-op> o => Op(o', d)

  mut3:
    Expr(Infix(Op(s, d), x, Infix(o, e1, e2))) ->
    C([tdecl],
      [Expr(Infix(Op("=", AnyFun), t, Infix(o, e1, e2))),
       Expr(Infix(Op(s, d), x, t))])
    where <is-mutating-op> s;
          <make-temporary> o => (t, tdecl)

  find-mutating-op:
    Op(s, d) -> Op(s', d')
    where <shadow-get> d => (d', u, ug);
          <fun-get-name;get-name-as-string> d' => s';
```

Fig. 9. Stratego source code for mutification

Rule `mut2b` is used in the case where `x` *is* used in `e2`. An extra temporary is used to preserve the value of `x` across the assignment of `e1` to `x`. All instances of `x` in `e2` are replaced by the new temporary variable.

The last rule, `mut3`, further decomposes an already mutified expression. If `x o1= e1 o2 e2` matches, a temporary is introduced to hold `e1 o2 e2`; this expression then becomes suitable for transformation by `mut2a` or `mut2b`. In this way, all occurrences of non-updating operators are removed, and replaced by updating operators.

The definition of `find-mutating-op` is also shown. It uses a simple table look-up (with `shadow-get`) to map from non-updating to updating operators.

A further improvement is possible: if we allow the user to specify the mathematical properties of operators, we can take advantage of commutativity, and do `x = e op x` in just a single step: `x op= e`.

Used together with the CodeBoost transformation library, the code in Fig. 9 is a complete implementation of the simplified mutification. The general case needs to handle mutification of functions in addition to operators, and general expressions for arguments instead of just simple variables or binary operator expressions.

5 Conclusion

In this paper, we have presented a framework for the source-to-source transformation of C++ programs. The implementation of such source-to-source transformations requires a considerable language processing infrastructure capable of parsing and pretty-printing programs, performing semantic analysis and implementing the transformations themselves. In particular, for a complex language such as C++ [12] this requires a large effort.

The CodeBoost framework provides the basic infrastructure needed to apply Stratego transformations to C++ programs. CodeBoost provides a parser and pretty-printer for a considerable subset of C++, ensuring the production of syntactically correct C++ programs. CodeBoost performs semantic analysis, and supports function and operator overloading, and templates. 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 has been developed as part of the SAGA project, to support the Sophus style of programming [11]. Sophus is a C++ library providing high-level abstractions for implementing partial differential equation solvers. CodeBoost is used to reduce the overhead caused by the abstract implementation style and to implement optimisations that use knowledge of the mathematics behind the library functions. CodeBoost has also been used to implement a path profiler for coverage analysis.

Program transformations are often implemented using ad hoc tools. One of the most commonly used basis for such tools is the general-purpose string

processing language Perl [17]. Although Perl is a powerful language, and has excellent lexical rewriting features, it offers little support for syntactic analysis (or even representing a syntax tree) and is not usable for complex transformations. Nevertheless, Perl has proven useful in pre- and postprocessing stages in CodeBoost.

The algebraic specification formalism ASF+SDF [8] has strong syntactic capabilities, and supports rewriting with high-level transformation rules. It was used in a first experimental version of CodeBoost [9]. 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.

OpenC++[5, 6] is an object-oriented C++ transformation tool. It satisfies most of our requirements, but transformations are specified in low-level (from our point of view) C++, declarator parsing is incomplete, and it does not support cascading transformations. OpenC++ operates on a concrete syntax tree, and the output of a transformation is simply a string representation of C++. For cascading transformations, the output string would have to be reparsed, but due to the ambiguous nature of the C++ grammar, parsing a code fragment without its surrounding context is next to impossible. This would also be a problem for ASF+SDF, where rewrite rules are specified in concrete syntax.

Frameworks with similar aims for transformation of other languages exist. An example is Compost [1], a transformation framework for Java written in Java.

For numerical software the TAMPR program transformation system [3, 4] 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.

So far, CodeBoost has been used for two projects; the implementation of a path profiler, and the implementation of experimental domain-specific optimisations for the Sophus project.

The development of CodeBoost is ongoing, and it will be expanded to cover a larger part of the C++ language and with more program transformations. Further, we are developing a larger base of examples showing the versatility and usefulness of such a tool.

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

References

[1] Uwe Assmann. COMPOST. The software composition system. http://i44www.info.uni-karlsruhe.de/~compost/.

[2] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Paris, France, 1996.

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

[4] James M. Boyle, T.J. Harmer, and V.L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, Boston, 1997.

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

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

[7] Merijn de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, University of Wollongong, Australia, 2000.

[8] A. van 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.

[9] T.B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*.

[10] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[11] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

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

[13] 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.

[14] Mark van den Brand and Merijn de Jonge. Printing within the ASF+SDF Meta-Environment: a generic approach. Technical Report SEN-R9911, CWI, Amsterdam, The Netherlands, 1999.

[15] Eelco Visser. Language independent traversals for program transformation. In Johan Jeuring, editor, *Workshop on Generic Programming (WGP2000)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[16] 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).

[17] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition, 1996.