Electronic Notes in Theoretical Computer Science 86 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume86.html 15 pages

Domain-Specific Optimisation with User-Defined Rules in CodeBoost

Otto Skrove Bagge [1]

Chr. Michelsen Research AS
PO Box 6031, N-5892 Bergen, Norway

Magne Haveraaen [2]

Department of Informatics
University of Bergen
PO Box 7800, N-5020 Bergen, Norway

Abstract

The use of domain-specific optimisations can significantly enhance the performance of high-level programs. However, current programming languages have poor support for specifying such optimisations. In this paper, we introduce user-defined rules in CodeBoost. CodeBoost is a tool for source-to-source transformation of C++ programs. With CodeBoost, domain-specific optimisations can be specified as rewrite rules in C++-like syntax, within the C++ program, together with the classes where they apply. We also illustrate the effectiveness of user-defined rules and domain-specific optimisation on a real-world example application.

1 Introduction

High-level, abstract programs communicate clearly the intent of the programmer. Much of this information is lost upon translation to a lower-level representation—either by the programmer or by the compiler. Traditional optimisation techniques focus much attention on various forms of analysis to rediscover this lost information. *Domain-specific optimisations* take advantage of domain knowledge for optimisation purposes. This allows the optimiser to optimise high-level code directly, with little need for analysis. This greatly simplifies implementation, and also makes it possible to implement otherwise infeasible special-purpose optimisations. However, it also places additional

[1] Email: otto@codeboost.org
[2] Email: magne@ii.uib.no

demands on the programmer, who must formalise the knowledge of the abstractions used, and make this information available to the optimiser.

This paper describes user-defined rules in CodeBoost, which provide one way of formalising domain knowledge for use in optimisations.

CodeBoost [1,2,9] is a source-to-source transformation tool for C++ [13,20]. It has been developed as part of the SAGA project, to serve as a domain-specific optimiser for Sophus [12,16]. Sophus is a C++ library providing high-level abstractions for implementing partial differential equation solvers.

CodeBoost provides a framework for implementing domain-specific optimisations and other transformations. It contains a parser, a semantic analyser, a transformation library, and a pretty-printer. CodeBoost is implemented in the Stratego program transformation language [22,23], and new transformations may be implemented either in Stratego, or as user-defined rules embedded in a C++ program.

User-defined rules allow C++ programmers to specify domain-specific optimisations and other transformations within the C++ program. Rules are specified in C++-like syntax, eliminating the need to learn a separate transformation language. Because CodeBoost performs semantic analysis also on user-defined rules, it is trivial to perform semantic as well as syntactic matching.

The purpose of this paper is to give an overview of the capabilities and use of user-defined rules in CodeBoost. We will first introduce the basic ideas and concepts of user-defined rules. We then discuss some slightly more advanced concepts: list matching, conditions and generic rules. Then, we illustrate the practical use of CodeBoost on a real-world Sophus application, with a few simple, yet effective rules. Finally, we outline our future plans, discuss related work and offer some concluding remarks.

2 User-defined rules

User-defined rules in CodeBoost allow C++ programmers to specify rewriting of expressions. They are primarily intended as an aid in writing domain-specific optimisations, but they are useful also for other kinds of transformations which require systematic rewriting of expressions. Examples include replacing all uses of unsafe indexing operators in a library with boundary checking versions, or instrumenting code with extra checks to verify the validity of optimisation rules.

Furthermore, user-defined rules can be used to express function relationships and properties that may otherwise only be explicitly available in formal program specifications or design documents. Having this kind of information available as rules may be useful for both documentation, optimisation and testing purposes.

Since rules are specified in C++-like syntax, there is no need to learn a whole new programming language, nor is knowledge of compiler design or

the inner workings of an optimiser required to write user-defined optimisation rules. To improve maintainability, the rules can be placed within the C++ program, close to relevant functions and classes.

2.1 Semantic matching

In CodeBoost, all program transformations are performed on an abstract syntax tree (AST) representation of the program. To support advanced optimisations, the AST is annotated with semantic information. For instance, all variables are annotated with types, and all function calls are annotated with the unique function signature corresponding to the called function. This makes it easy to distinguish between overloaded functions. [3]

The ability to do matching on syntactic as well as semantic information is important when implementing domain-specific optimisations for a language which allows overloading. For instance, suppose that part of the domain knowledge for a numerical library is that

$$\texttt{pow}(\texttt{x}, 2) \equiv \texttt{x} * \texttt{x}. \tag{1}$$

There may be other unrelated `pow` functions for which this is not true. Any optimisation rule taking advantage of (1) must not interfere with such unrelated functions.

Convenient matching on semantic information is not as simple as it may seem. Explicitly specifying function signatures and variable types in rules can be tedious and error-prone, particularly since the rules must often be kept consistent with application or library code. Secondly, since CodeBoost does not support the use of concrete C++ syntax in Stratego rules, rules must be written in the rather verbose abstract syntax. Matching a single call to a particular function may require an abstract syntax pattern several lines long, making reading and writing such rules a daunting task.

In CodeBoost, we have solved these problems by allowing rules to be specified within C++ programs, with patterns written in C++ syntax. When the program is run through CodeBoost, the patterns are converted to AST form and subjected to normal semantic analysis, together with the rest of the program. Hence, they are automatically annotated with the correct signatures. After semantic analysis, the rules are extracted and made available for use as rewrite rules in CodeBoost modules.

2.2 Rule syntax

All user-defined rules are placed inside one or more C++ functions named `rules()`. CodeBoost will recognise the function signature and interpret the body as a list of rules. The `rules()` definitions can occur anywhere a normal

[3] C++ allows overloading of both functions and operators. The distinction between functions and operators is often of little importance to CodeBoost, and we will usually refer to both as simply 'functions'.

function can, and contain any number of rules. All names used in the rules should be in scope and accessible. To simplify maintenance, rules are best placed close to the functions to which they apply (i.e. inside the same class).

A user-defined rule consists of a *rule name* , a *match pattern*, a *replacement pattern* and an optional *condition*. Fig. 1 shows the syntax for user-defined rules. A typical rule looks like this:

```
void rules()
{
  int x;
  simplify: pow(x, 2) = x * x, isTrivial(x);
}
```

where '`simplify`' is the rule name, '`pow(x, 2)`' is the match pattern, '`x * x`' is the replacement pattern, and '`isTrivial(x)`' is a condition. During semantic analysis the `pow` and `*` calls will be annotated with signatures, ensuring that the correct versions are used for both matching and replacement. The hypothetical condition `isTrivial` would check that `x` is a trivial expression, so that the rule does not cause work duplication.

The rule name implicitly controls the application strategy. For instance, the `simplify` transformation module will apply `simplify` rules according to a predefined topdown-repeat strategy. This is explained in more detail in Section 2.3.

When a rule is applied to an expression, the AST structure of the match pattern is compared to the AST structure of the expression. If the pattern matches, the condition is checked; if it is true, rule application succeeds and the matched expression is replaced with the replacement pattern.

Locally declared variables (such as the `x` above) are treated as meta-variables. A meta-variable in the match pattern matches all expressions, and is bound to the first expression it matches. Subsequent occurrences of a bound meta-variable in the match or replacement pattern are replaced by its value. For example, the match pattern `f(x, x)`, where `x` is a meta-variable, will match a call to `f` where both arguments are identical.

rules ::= `void rules() {` (*vardecl* | *rule*)* `}`

rule ::= *rule-name* `:` *expr* `=` *expr* [`,` *condition*] `;`

Fig. 1. Syntax for user-defined rules

2.3 Rule sets and application strategies

All rules with the same name make up a *rule set*. When a rule set is applied, all rules in the set are tried until one of them succeeds. If no rule succeeds, the rule set application fails. A rule set is said to terminate if it can be applied

repeatedly without getting stuck in an endless loop (i.e. it is guaranteed to fail sooner or later).

Rule set application is ultimately controlled by transformation modules written in Stratego. CodeBoost makes rule sets available to Stratego programs as ordinary Stratego rules. User-defined rules are typically used with one of two transformation modules, `simplify` and `apply-user-rules`, whose only purpose is to apply rule sets. With these two modules, users can write and apply their own rewrite rules without any knowledge of Stratego.

The rule sets applied by the transformation modules all have predefined names. However, the user is free to apply other rule sets in rule conditions (see Section 3.2).

The `simplify` rule set is used for simplification rules. It is assumed that the right-hand side of `simplify` rules is somehow "better" or preferable to the left-hand side, and that the rule set will terminate[4]. The transformation module `simplify` will apply the `simplify` rule set repeatedly (until it terminates) using top-down traversal.

The module `apply-user-rules` applies four rule sets. It uses a down-up traversal, applying `topdown` rules on the way down, and `bottomup` on the way up. For repeated application (as in `simplify`) there is `topdown_r` and `bottomup_r`.

In addition to these general purpose rule sets, some rule sets are used for specific purposes. For instance, some transformation modules use the `assoc` and `commute` rule sets to implement matching modulo associativity or commutativity.

3 Advanced Concepts

3.1 List Matching

When working with functions that accept a variable number of arguments, it is useful to be able to match all or part of the argument list with a single meta-variable. List matching makes this possible; the match pattern `_list_(x)` will match zero or more arguments in an argument list. When substituting meta-variables in the replacement pattern, the value of `_list_(x)` will be integrated into the argument list it appears in. For example, consider the rule

```
int f(int, int, ...);
int g(...);

void rules()
{ int a, b, c;
  topdown:  f(a, b, _list_(c)) = g(a, _list_(c), b);
}
```

[4] So, if the user specifies a rule that does not terminate, rule application may not terminate.

If this rule is applied to `f(1, 2, 3, 4, 5)`, it will match with `_list_(c)` ↦ `3, 4, 5`, and the result of the rewriting will be `g(1, 3, 4, 5, 2)`.

During overload resolution, `_list_(x)` will appear as a single argument with the same type as `x`. Note, however, that *matching* is untyped, so `_list_(x)` can match arguments of any type—the type information is only used for overload resolution. This is consistent with the semantics of C++.

It is possible to use list matching to match the beginning or middle of argument lists:

```
topdown: f(_list_(a), g(b), _list_(c))
         = fg(b, _list_(a), _list_(c));
```

If this rule is applied to `f(1, 2, 3, g(4), 5)`, it will match, with `_list_(a)` ↦ `1, 2, 3`; `b` ↦ `4`; `_list_(c)` ↦ `5`; and the result will be `fg(4, 1, 2, 3, 5)`.

CodeBoost only looks ahead one element while deciding when to stop, and no backtracking is done. If the list match pattern is followed by something that matches anything (such as an unbound meta-variable), the list match will never match anything.

3.2 Rules with Conditions

Conditions are useful for specifying that a rule should only be applied in certain cases. Because conditions can have (local) side-effects, such as binding new meta-variables, they can also be useful for specifying more advanced rewriting. Conditions follow the rule body after a comma, and are written in function call notation. The comma should be read as 'where'. For example:

```
X x, y;
simplify: (x + y) = x, isZero(y);
```

In this rule, `isZero` is a hypothetical built-in condition, which uses dataflow information to determine whether `y` has a zero value. The rule will only be applied if the condition evaluates successfully.

Built-in conditions are available for doing advanced checking and rewriting. So far, the development of the condition language (and user-defined rules in general) has been driven by the needs of particular Sophus optimisations rules. Therefore, only a few special built-ins have been added so far. Conditions can be combined using several primitives, including `&&` (and), `||` (or) and `not`.

As an example, the following rule is a better version of the `pow`-simplification rule from Section 2.2:

```
int x, t;
simplify: pow(x, 2) = t * t, t = tmp(x);
```

The rule uses the `tmp` built-in to make a temporary variable to hold the value of `x`, so that `x` is not evaluated twice. A declaration for the temporary is inserted before the containing statement, and `tmp` yields the name of the temporary, for use in the replacement.

In addition to the built-ins, all user-defined rules can be used as conditions. The following rule will switch the arguments of the plus operator so that multiplications are on the left side. If there already is a multiplication on the left side, nothing will be done.

```
topdown: z + (x * y) = (x * y) + z,
                    not(isMultExpr(z));
isMultExpr: (x * y) = true;
```

3.3 Generic Rules

So far, we have only written rules that work on *specific* functions and operators. Many rules have a basic structure that is independent of the particular functions to which they apply. For example, the structure of a commutativity rule is the same no matter which operators or types are involved (i.e., switch the two arguments of a binary function or operator). Writing the same rule over and over again quickly becomes tedious. It is better to write a generic rule, and then specify which operators are commutative—particularly since other rules may also depend on the commutativity property.

A *generic rule* can be used to specify rewrite rules independent of names, types and signatures. For example, if `f` is declared locally as a function pointer, the generic rule

```
topdown: f(x, y) = f(g(x), g(y));
```

will rewrite *any* call to *any* binary function or operator. The `f` will be bound to both the name and signature of the matched function, and can be used to construct new calls to the same function in the right-hand side of the rule.

As another example, the following fragment gives a generic commutativity rule:

```
void rules()
{
  T (*f)(T, T);  // declare f as function pointer
  T x, y;

  commute: f(x, y) = f(y, x),
                  commutative(!f(x, y));
}
```

The `!` primitive is used to build a literal expression term for use in rule application. The following rules specify that the operators `*` and `+` on integers are commutative:

```
void rules()
{
  int a, b;
```

```
  commutative: (a + b) = true;
  commutative: (a * b) = true;
}
```

Obviously, for simple cases like the above example, generic rules are not a huge benefit. However, they are beneficial for more complicated and non-obvious rules. Furthermore, generic rules provide the benefit of abstraction; the rule and its requirements are separated from, and independent of, the concrete functions that fulfil its requirements.

4 Application: Sophus

4.1 Sophus background

We will use the Sophus software library [10,12] as our test bed for transformations. Sophus is developed for the numerical solution of partial differential equations. Such programs are often termed *solvers*. Solvers are often used to check whether a model of the real world, e.g., a geophysical model of a 1km^3 section of the earth, matches the real world which it is supposed to model. A check is performed by letting the solver simulate some measurement on the model, e.g., how seismic waves propagate, and compare the simulated results with real world measurements. If the comparison shows a problem with the model, the model is adjusted and the solver is run once more on the new model. Comparing results and adjusting the model is a creative process which requires human ingenuity and may easily take specialists a full working day. A simulation cycle is such a sequence of activities (run solver, investigate results, adjust parameters for the next run of the solver). In this field runtime efficiency is important. A solver may easily take several hours or days to reach a solution. Bringing the solution time down from say, 25 hours to 15 hours makes it possible to complete a simulation cycle every day rather than every 2–3 days.

Unlike most numerical libraries, Sophus is based on coordinate-free notions, which are very high-level abstractions. This makes Sophus well suited for source-level optimisations. We will focus on the SeisMod application, which simulates seismic waves (elastic waves) in geophysical models of sections of the Earth.

A key component of SeisMod is the Mesh abstraction, akin to the notion of an array. Meshes are used to store data about the waves and the physical properties of the material where they propagate. If we are doing a coarse simulation on a 1km^3 section of the Earth, we only need to store data with 5m resolution, which requires $200^3 = 8\,000\,000$ units of information. The information needed for a simulation may easily amount to 20 Mesh variables, each with 8 000 000 floating point numbers.

It is not always necessary to simulate a full 3D section. In many cases a 2D cross section of the model will give sufficient information. In our example

this reduces storage (and computation requirements) for a 2D version by a factor of 200, to less than one percent of that of the 3D version.

4.2 Mesh, MeshPoint and MeshShape abstractions

A consequence of working with both 2D and 3D models is that we sometimes need to index meshes using three indices (3D case), other times we will need only two indices (2D case) for the same Mesh variables. In Sophus this has been solved by introducing an index type abstraction called a MeshPoint, which represents a list of indices with a given shape. The MeshShape abstraction tells how many indices are used, and the extent of each of them. In the example, the MeshShape would be $\langle 200, 200, 200\rangle$ and $\langle 200, 200\rangle$ in the 3D and 2D cases, respectively. So, a Mesh M is indexed by $M[P]$, where the MeshPoint P may be a list of 2 or 3 indices. Actually, there are also cases where we need only one index, in others we need four or more.

For this paper, the mesh abstractions contain a few interesting operations:

`float operator[](const Mesh &, const MeshPoint &)` which is the Mesh indexing function, returning a floating point value for every appropriate MeshPoint.

`int getlex(const MeshPoint &)` which decodes a MeshPoint index to the unique lexicographic ordering it has within its shape.

`MeshShape getshape(const MeshPoint &)` which extracts the shape of a MeshPoint.

`int getsize(const MeshShape &)` which computes the number of distinct MeshPoints with the given shape.

`MeshPoint setlex(const MeshShape &, const int &)` which encodes an integer as the unique MeshPoint with that lexicographic ordering within the given MeshShape.

Obviously the operations `getlex` and `setlex` are inverses, in the sense that `getlex(setlex(S,i))==i` and that `setlex(getshape(P),getlex(P))==P` for MeshShape `S`, integer `i` and MeshPoint `P`. These properties of the MeshPoint abstraction may easily be encoded as user-defined simplification rules within the MeshPoint class:

```
void rules()
{ int i;
  MeshShape S;
  MeshPoint P;

  simplify: getlex(setlex(S,i)) = i;
  simplify: setlex(getshape(P),getlex(P)) = P;
}
```

A typical implementation of a Mesh class will represent the data of a Mesh

with MeshShape `S` as an array `float A[getsize(S)];`. Placing the data in the Mesh in a lexicographic ordering, we can implement the indexing operation using the `getlex` function.

```
float operator[](const Mesh & M, const MeshPoint & P)
{ return M.A[getlex(P)];
}
```

A call `M[P]` computes the lexicographic position of the argument `P`, and then uses this integer to access the data in the array `A`.

4.3 Example: using user-defined rules for simplification

When using high level abstractions, it is often the case that the functions have to decode its data structure, do some computation, and encode the result within the data structure of the abstraction. Sometimes a succession of calls to such functions will result in encode-decode sequences which eliminate each other (at run-time). Often this code cannot be removed from the program text without breaking the abstraction barriers. One example is the MeshPoint abstraction, with a computation `MeshPoint P=setlex(S,i); float r=M[P];` for a Mesh `M` of MeshShape `S`. As a computation this first encodes the integer `i` as a MeshPoint, then decodes the MeshPoint back to the integer value of `i` in order to access the data of the Mesh. But we cannot eliminate this extra computation without revealing the data structure and algorithms we are using for implementing the Mesh. Further, bypassing the MeshPoint abstraction will force problems in other parts of the code, such as the partial derivative functions (not further discussed here), where the list nature of the MeshPoint index is important.

In the SeisMod solver much of the computation is centred around traversing the Mesh data and performing operations on the elements of several meshes for every MeshPoint index. Since the MeshShape `S` for the MeshPoints is not known until runtime, traversal of the Mesh `M1` ... `Mn` data structures are typically done using a simple loop and the `setlex` function.

```
for (int i=0; i<getsize(S); i++)
{ P = setlex(S,i);
  ... M1[P] ... Mn[P] ...;
}
```

For every iteration of the loop, we encode i to a MeshPoint, then every indexing operation decodes the MeshPoint back to the same integer.

A code transformation tool like CodeBoost is allowed to bypass the abstraction borders of the code, e.g., by inlining, and may thus exploit properties of the underlying implementation. In our case the user-defined rule `getlex(setlex(S,i)) = i` allows CodeBoost to get rid of the unneeded computations. The effect of this optimisation on the SeisMod code is startling. Table 1 summarises timings for the isotropic version of SeisMod, baseline ver-

sion 1.21, with the accompanying small and large regression testing data sets.

	Not optimised	*Basic*	*Simplified*	*Speedup*
Small	827s	630s	111s	5.7
Large	25435s	19028s	3996s	4.8

Table 1
Timings for SeisMod for small and large data sets. *Not optimised* is results without any CodeBoost optimisation. *Basic* includes some Sophus-specific optimisations. *Simplified* adds inlining of indexing and `getlex`/`setlex` simplification. *Speedup* is speedup factor of *Simplified* relative to *Basic*.

5 Future Work

Although user-defined rules have already been used with great success in optimising Sophus applications, they are still an experimental feature under active development. Currently, our work is focused on the interaction between abstraction and optimisation, and the ordering of transformations. Many domain-specific optimisations are applicable only at a particular abstraction level. Inlining can be used to lower abstraction levels, but it must be done at precisely the right time, or optimisation opportunities may be lost.

Transformation rules can be classified into three different kinds: *Simplifying rules*, which perform actual optimisation, *Implementing rules*, which perform inlining, and rules such as commutativity, which do not change the abstraction level, and do not result in better code, but may enable other optimisations. A simple strategy exploiting this classification is: 1) Apply as many simplifications as possible; 2) Try other rules, to see if they enable further simplification; 3) Apply one level of inlining, go back to 1 if this succeeds. But even with strategies such this, there is still the question of choosing which individual rule to apply.

Sometimes, more than one rule matches a given expression. In this case, CodeBoost will currently pick the first rule. It would probably be better to pick the most specific rule (i.e. the one with fewest meta-variables), or the one that gives the "best" results. Ordering rules by specificity is easy, but deciding which rule is "best" will probably require input from the user. Furthermore, a rule that may seem "worse" could still enable other transformations resulting in overall better results. We have not studied this in detail yet; it may be feasible to try several possibilities in parallel and then pick the best result.

6 Related Work

The concept of conditional rewrite rules is pretty standard in transformation languages such as Stratego [22,23], ASF+SDF [8], ELAN [3], TXL [7],

TAMPR [4,5] and others. Unlike these systems, our user-defined rules facility is not intended as a general-purpose transformation language; we restrict ourselves to transforming C++, within the CodeBoost framework. Apart from this, the main difference compared to general-purpose systems is the embedding of rules within the program being transformed. This allows users to place domain-specific optimisation rules inside the modules to which they apply, and also allows the use of normal semantic analysis to generate semantic constraints.

Embedding optimisation rules in program text is not a new idea. We were first inspired by the rewrite rules in the Glasgow Haskell Compiler [14]. Our implementation is more advanced, however, as it supports both side conditions and multiple strategies.

User-definable strategies is available in Stratego and ELAN, and gives the programmer precise control over the application of rules. We offer only a selection of predefined strategies; however, when user-defined rules are used from Stratego, they can of course be applied using arbitrary strategies.

User-defined rules bear some resemblance to macro processing, as in Lisp [19] and the rather primitive C/C++ preprocessor [15,20]. However, macros are commonly used for *syntactic rewriting*, without taking into account context or semantics (this is actually a useful feature in some cases). The ability to match on semantic information is an important part of user-defined rules in CodeBoost.

CodeBoost, and user-defined rules, is most appropriately compared with similar frameworks for C++ and other languages. Simplicissimus [18] is a transformation system for C++ that allows user-specified conditional rewriting of expressions. Pattern matching is done with *expression templates* [21], and *traits* [17] are used to specify function properties, which can be used in rule conditions. 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).

Simplicissimus is implemented as a compiler plug-in (currently using GCC), and uses the compiler's template processing for matching. The functionality is similar to that of user-defined rules. The main drawback with Simplicissimus is that the syntax for specifying rules is verbose and complex; rules that are written in one line using user-defined rules would need half a page of code in Simplicissimus. Our generic rules are inspired by Simplicissimus, and the ideas of arbiters, directors and stages will probably be useful in the further development of CodeBoost.

OpenC++ [6] 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. OpenC++ is

not restricted to working with expressions; it can also be used for language extensions, and generating functions and classes.

The Broadway compiler [11] allows library designers to annotate their C++ 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 equations 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.

For numerical software the TAMPR program transformation system [4,5] has been used with remarkable success. A typical use is the derivation of efficient Fortran code from high-level functional specifications. Other uses include deriving an efficient implementation of TAMPR itself (which is specified in pure functional Lisp), reverse engineering, and solving the Year 2000 problem.

7 Conclusion

User-defined rewrite rules provide a convenient way of specifying domain-specific optimisations. Rules are written in C++-like syntax within the program text, and allow conditional rewriting of expressions. Conditions can be used both to control when and if a rule is applied, and to perform advanced rewriting by calling other rules or calling built-ins written in Stratego. Additionally, list matching is available to conveniently manipulate the argument lists of functions accepting a variable number of arguments.

Specifying rules within the program text allows us to subject rule patterns to semantic analysis together with the rest of the program. Automatically adding semantic information to rule patterns enables simple and easy specification of domain-specific optimisations, without the need for hand-written semantic constraints.

User-defined rules are only a small part of the CodeBoost framework, and more complex, generic optimisations are best implemented as separate transformation modules written in Stratego. Still, as shown in this paper, we have promising results from using user-defined rules for optimising Sophus, and we expect that most new Sophus optimisations will be developed using user-defined rules.

8 Acknowledgements

Many thanks to May-Lill Sande, Karl Trygve Kalleberg, Eelco Visser and the anonymous referees for useful comments and inspiration. This investigation has been carried out with the support of the Research Council of Norway

(NFR), and by a grant of computing resources from NFR's Supercomputer Committee.

References

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

[2] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press. (To appear).

[3] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In C. and H. Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, Pont-A-Mousson, France, September 1998. Elsevier Science.

[4] 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.

[5] 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.

[6] 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.

[7] James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings of the IEEE 1988 International Conference on Computer Languages*, pages 280–285, October 1988.

[8] Arie van Deursen, Jan Heering, and Paul 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*, 8(4):247–259, 2000.

[10] Helmer André Friis, Tor Arne Johansen, Magne Haveraaen, Hans Munthe-Kaas, and Åsmund Drottning. Use of coordinate free numerics in elastic wave simulation. *Applied Numerical Mathematics*, 39(2):151–171, 2001.

[11] Samuel Z. Guyer and Calvin 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.

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

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

[14] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001.

[15] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

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

[17] Nathan C Myers. Traits: a new and usefule template technique. *C++ Report*, June 1995.

[18] Sibylle Schupp, Douglas P. Gregor, David R. Musser, and Shin-Ming Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, October 2002.

[19] Guy L. Steele, Jr. *Common LISP: The Language*. Digital Press, second edition, 1990.

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

[21] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[22] Eelco 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.

[23] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew 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.