

Library Transformations

Sibylle Schupp, Douglas Gregor, David Musser
Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York 12180
{schupp,gregod,musser}@cs.rpi.edu

Shin-Ming Liu
Hewlett-Packard
California
shin@cup.hp.com

Abstract

*While software methodology encourages the use of libraries and advocates architectures of layered libraries, in practice the composition of libraries is not always seamless and the combination of two well-designed libraries not necessarily well designed, since it could result in suboptimal call sequences, lost functionality, or avoidable overhead. In this paper we introduce *Simplicissimus*, a framework for rewrite-based source code transformations, that allows for code replacement in a systematic and safe manner. We discuss the design and implementation of the framework and illustrate its functionality with applications in several areas. *Simplicissimus* is integrated into the Gnu C++ compiler.*

1. Introduction

In the process of software development, software libraries have now become widely used. Many programmers have come to realize that the use of libraries results in code that is more robust and reliable than what they could have developed on their own from scratch. Even library designers themselves apply the principles of reusability to their software production and begin to stack libraries, e.g., to develop a graph library on top of a simple container library or a linear algebra library on top of an array library. The idea in all cases is to combine two or more independently developed library components and to plug them together in some particular way. In practice, however, the compositions are not always as seamless as one would wish, even in libraries of parameterized components that are explicitly designed with plug-in capabilities. Of much subtlety are problems that do not manifest themselves in visible program errors, but in the form of suboptimal call sequences, lost functionality, or avoidable overhead.

Scripting languages have become popular as they provide quick ways to glue together software pieces that do not otherwise fit. But, the glue mechanisms are based on

little syntactic and no semantic analysis and therefore are restricted in their applicability; where they apply, they replace the original source code in an essentially unchecked manner. In contrast, we advocate a tool that can handle source code to its full extent—not just a subset of it—and take any two libraries and fix the semantic glitches of their composition in a systematic and safe way. In this paper we introduce *Simplicissimus* as a framework that meets these requirements.

Drawing from ideas in program transformation and term or expression rewriting, *Simplicissimus* solves library-interopability problems by representing the desired change as a rewrite rule where the left and right hand side of a rule represent the expression and its replacement, respectively. *Simplicissimus* provides a framework for the specification of these rewrite rules, their validated application, and their execution. It is an extensible, highly parameterized, and semi-automated module that depends on initial user information about the correctness conditions of a rewrite rule, the operands and operators it is applied to, but can then automatically select, validate, and apply the appropriate (sequence of) rules. The semantic checks are based on conceptual descriptions of formal and actual constituents of a rewrite rule on the one hand, and internally available concept hierarchies on the other hand. To ensure that the semantic checks are performed at compile time, *Simplicissimus* interfaces with the translating compiler.

We begin our presentation with a characterization of the problems that could occur in particular in the composition of parameterized libraries and give some evidence why these problems cannot be solved at the level of the libraries themselves, through redesign (sect. 2). Turning then to *Simplicissimus*, we describe its features in a language-independent way in sect. 3 and give some applications in sect. 4. The overall architecture, including the interface to the Gnu compiler, its implementation in C++, and the user interface are subject of sections 5, 6, and 7. We conclude the presentation with a discussion of related work in sect. 8.

Throughout the paper we describe *Simplicissimus*

mainly as a module for the transformation of libraries. Technically, however, there is nothing that would prohibit its use by application programmers: any user who is willing to provide the extra input, at the requested conceptual level, could use the framework. The main reason we address library designers is that we can assume them not only to have the required domain expertise but also to accept the initial extra effort that the preparation of the requested input requires, since it results in an improved service for each client of their library.

2. Library interoperability problems

If two libraries are well designed, shouldn't their composition be well designed too? If a parameterized library, when instantiated, loses some of its properties, doesn't that indicate flaws in its design? To give a first idea of the nature of the problems that can occur in the composition of libraries we contrast *design by parameterization* with the most widespread alternative for library development, *design by inheritance*. In the inheritance-based approach a new library is constructed as the refinement of an existing library, where new classes are derived from existing ones by adding new data type invariants or extending their functionality. The base library has to be organized, typically through abstract functions, so that certain constraints get propagated to the newly derived classes. Aside from that, however, it is "embedded" in the derived library and, unaffected by this embedding, continues to work as-is. In contrast, the behavior of a parameterized library varies depending on the way its type parameters are bound. In a parameterized approach much of the code of the *carrier library* is stated in terms of parameters and new libraries are constructed by first adding new layers of parameterization and then connecting the carrier library through (partial) bindings of its parameters. While a plug-in philosophy and the resulting loose coupling encourages the design of modular components that are mutually orthogonal, it obviously cannot guarantee that all components are based on the same major design decisions or are designed with the same design principles in mind. It is thus possible to plug together libraries with conflicting design principles, and this is exactly the situation where features of a library may get lost. In the remainder of this section we elaborate on the three major sources of library incompatibilities we have identified thus far.

First, many library designers give efficiency highest priority. They fear that their library will not be used if it cannot keep up with the most efficient, probably non-parameterized, implementation elsewhere and, with good reason, are concerned that all relative performance losses weigh even heavier if their library constitutes the foundation for applications on top of it. If given a choice in the

libraries they use themselves, they therefore choose a fast implementation over a safe one. A standard example of the resulting problems their users then face are safe and unsafe vector (array) access functions. While for example the Standard Template Library (STL) provides both functions, a fast subscript operator on the one hand and the `at` member function, which does bounds checking, on the other hand, most high-performance libraries on top of it use the former function only and implement their access functions in terms of the fast but unsafe subscript operator. If, however, a user would have preferred safety checks over speed, (s)he currently simply has bad luck. Using *Simplicissimus*, however, users can overrule the priorities of the library designer and temporarily replace, for example, the unsafe vector access function by the one that performs a boundary check.

Second, based on the same priority for efficiency, the readability of code suffers from optimizations at the source code level. Especially in the area of numeric libraries where Fortran code still reigns, library designers have developed techniques that, anticipating optimization opportunities for compilers, result in code that outperforms Fortran code [24, 29] but completely breaks with mathematical notational conventions. After the first euphoria about the performance results of these libraries users now are divided into two camps: those who value efficiency above all and those willing to sacrifice some of the efficiency gains in lieu of better readability and lesser maintenance costs. Again, library designers cannot serve both needs: they either apply or do not apply their optimization techniques. And again, *Simplicissimus* can be used to conciliate two contradictory demands: when the optimizations do not take place directly in the source code, but are represented as rewrite rules and applied separately, it is possible to follow user conventions and to provide readable code without any performance loss.

The genericity itself, lastly, which is inherent in parameterized libraries, can cause problems due to the fact that the carrier library and its instantiation have been designed without any knowledge of each other. An instantiating library might follow different interface conventions than the carrier library, e.g., different naming conventions or different default bindings for arguments. Moreover, it might provide specialized functions that do not get used because the carrier library fails to invoke them. In practice, therefore, it frequently happens that optimizations get lost and that the composition of two high-performance libraries results in a non-competitive library. The problem here is that it is difficult for the carrier library to be prepared for special functions the existence of which is not yet known. With *Simplicissimus*, however, the library to be plugged-in no longer depends on the preparations by the carrier library but can set up rewrite instructions to redirect an invocation to one of its own special functions.

$x + 0$	$\rightarrow x$	$x : T, (T, +, 0) \in \text{Monoid}$
$x*(y + z)$	$\rightarrow x*y + x*z$	$x, y, z : T, (T, +, *) \in \text{Ring}$
$x = y*z$	$\rightarrow \text{multiply}(x, y, z)$	$x, y, z : T, T \in \text{LiDIA}$
$x = T(0)$	$\rightarrow x.\text{assign_zero}()$	$x : T, T \in \text{LiDIA}$

Figure 1. Concept-based rules (left) and the requirements for their parameters (right)

3. Features

Simplicissimus, as already mentioned, is a framework for rewriting libraries at the expression level. That means that the manipulations are done through rewriting function invocations, including operator, function, and procedure expressions as well as global functions and member functions. Simplicissimus assumes instantiations that are syntactically correct but do not fully behave as intended, and restores the intended behavior by a sequence of expression transformations. In this section we discuss its major features of extensibility, safety, and manipulations at instantiation time, further explaining at the same time what expressions can be rewritten, under what circumstances, and when.

3.1. Extensibility

For a tool like Simplicissimus there is little that could be anticipated in the design of its rewrite rules or the types involved: it is solely the users who define which transformations are needed and where they should apply. It is therefore crucial to organize the framework so that it can be extended in various ways:

- by specifications of expressions that should be rewritten and the conditions that have to be met;
- by types, function, operators that may be rewritten;
- by strategies such as `first-fit` or `best-fit` that control the application of rewrite rules and their termination.

Since the framework is flexible enough to let users essentially define the whole system, it has to take care at the same time that the rewriting of source code preserves the original semantics. Even if, for example, replacing a function by a clone that just contains additional debugging information might be an innocent action, it still requires guarantees that, in spite of overloading, partial specialization, and other ambiguity-prone language features, the intended function is the only one that gets replaced. Even more critical, obviously, are semantic checks of rewrites that do not just annotate an existing body but exchange it for another one. One solution is to pass the responsibility for correct transformations back to the users. This, however, is not only unsafe but also difficult to implement efficiently: the more

dependence on the user, the less the translating compiler can enforce. To maximize the amount of checking that can be done automatically and at compile time, we therefore follow a concept-based approach.

3.2. Conceptual requirements

In Simplicissimus we state a rewriting rule at a conceptual level, in terms of the logical or algebraic properties on which it depends for correctness. More precisely, by “concept” we mean a (possibly infinite) set of abstractions that obey a (finite) set of logical or algebraic properties (requirements). Here we take the abstractions to be types, and we associate a rewrite rule with the concept with the minimal set of requirements that enable its correct application. Thus it can be correctly applied to expressions over every type in the concept, including not just those that might be known at the time the rule is designed but also any user-defined types that can be shown to belong to the concept (by showing all of the concept’s requirements are met). For example, the first of the rules shown in Fig. 1 is a right-identity simplification rule, which comes directly from one of the equations of the Monoid concept. Although stated in terms of $+$ and its identity element 0 , these are actually parameters, so that the rule also represents, for example,

$$\begin{aligned} x*1 &\rightarrow x & x : T, (T, *, 1) \in \text{Monoid} \\ x\&\text{true} &\rightarrow x & x : \text{bool}, (\text{bool}, \&, \text{true}) \in \text{Monoid} \end{aligned}$$

The second rule in Fig. 1 lists another algebraic example from the Ring concept and two examples of rules for a *LiDIA* concept. *LiDIA* [28] is a library for computational number theory for various multi-precision types, lattices, quadratic number fields, and polynomials. For efficiency it not only defines operators like $*$ on its numeric types but also functions that place their result directly in a parameter to avoid the creation of temporaries that can occur if the operator is used. Applications of the *LiDIA* rules are discussed further in sect. 4, where we describe the kinds of rules that library developers might profitably devise for use by Simplicissimus for optimizing uses of their libraries. Devising rules such as those shown here for the Monoid and Ring concepts would not be the province of library developers; rather, they are included in the base of algebraic concepts by which Simplicissimus organizes simpli-

fication rules for optimization of built-in and user-defined types [25].

Concepts are naturally arranged in a refinement hierarchy: for example, the Ring concept is a refinement of the Monoid concept, having more requirements and containing fewer types. In documenting C++ template classes and functions, concepts can be used to specify explicitly which requirements each type must satisfy in order to be used as a specialization of a certain template parameter [4, 11]. Closely related is the use of concept hierarchies in the design of generic algorithms; an algorithm is associated with the concept with the minimal requirements the algorithm needs for both correct and efficient operation. In *Simplicissimus*, by analogy, we seek to state a simplification rule at the level of the most general concept—the one with the fewest requirements—for which it is a correct transformation.

Our definition of concept derives mainly from the Tecton concept description language [13, 14, 21], in which the abstractions are algebras. There are also connections to formal concept analysis [31, 32], in which the lattice theoretic properties of concept hierarchies are emphasized, but in which instead of complex abstractions like types or algebras one typically has only simple objects. By organizing abstractions like types or algebras in a concept hierarchy one obtains a sound basis for design and organization of collections of other closely related abstractions—generic algorithms in the case of the generic programming paradigm, or, in our case, generic program transformation rules.

3.3. Instantiation-time manipulations

A third important feature of *Simplicissimus* is that all source code manipulations take place at instantiation time of a parameterized library. Instantiation time should thereby be seen in light of the two alternatives, run time and compilation time. Manipulations at run time, obviously, should be avoided for efficiency reasons, and we will show in sect. 6 how expressions are internally represented so that they can be evaluated symbolically and their transformation can take place before run time. Manipulations at compile time, on the other hand, would be too early for our purpose. For parameterized libraries both the need for a manipulation and its particular kind in part depend on the instantiating library, and thus cannot be determined before both libraries have been plugged together.

The requirement of instantiation-time manipulations is a distinctive feature of our approach (see sect. 8). It implies at the same time that the code to be rewritten and the instructions for its rewriting are jointly compiled. Sect. 5 further explains the single steps of the transformation of an expression and gives an overview of the overall architecture. Before turning to the implementation, however, we illustrate

with several examples how the framework can be used.

4. Applications

Simplicissimus started out as a project for an optimizing compiler, where we wanted to give users control over the well-known optimization of algebraic simplification—hence the name *Simplicissimus*—as well as the ability to “teach” the underlying compiler their own algebraic simplification rules [25]. It turned out, however, that the mechanisms implemented apply to source code manipulations in a much broader sense. To highlight possible applications, *Simplicissimus* now can be used

- to specialize functions and in particular optimize them;
- to extend functional specifications or to include additional safety features;
- to improve code readability, e.g., through organizing desugaring as rewriting;
- to annotate functions with debugging, tracing, or monitoring facilities.

In this section we illustrate some of the possible applications. We also list in each case how many lines of code users have to provide, so that reader can get a feeling for how little extra effort is required. For the interpretation of the number of lines of code, it is important to know that the requested information breaks down in three parts. Only the definition of a rule validator requires creative thought; the other two parts merely consist in filling out a trait class and wrapping up a constructor call. Sect. 7 discusses the details.

For the experiments of this section we chose practically relevant problems, some of which have been in the debate for some time. Although we ran the experiments in the context of particular libraries it should be clear how each experiment generalizes. The libraries used in the tests, all C++ template libraries, can briefly be characterized as follows:

- STL, the Standard Template Library, is a library of container types and generic data processing algorithms [1, 4, 20].
- MTL, the Matrix Template Library, is a library for various matrix types and fundamental linear algebra algorithms [26].
- LiDIA, as already mentioned, is a library for computational number theory [28].

Vector bounds checking Unlike safe languages like Pascal or Java, neither C nor C++ has array bounds-checking built-in. Since it is intended as a basis for added functionality, the STL vector class also omits bounds-checking in indexing expressions like `v[i]` to achieve maximum efficiency, and leaves it to the programmer’s responsibility to access valid vector elements only. Users can add bounds-checking only at the price of permanent source code modifications, either by manually replacing expressions such as `v[i]` by `v.at(i)` (since the vector `at` member function does do bounds checking) or, if they use one of several (non-standard) *safe vector* or *bounds-checking iterator* components that have been developed, by changing the variable declarations of vectors or iterators, or both. In either case, from then on, the original source code is no longer available. For this experiment we introduced a rewriting rule that advises the compiler to replace all original subscript operations by calls to the function `at`. Both the subscript operator and the `at` function have to be wrapped up and described in traits; the validator here comprises 10 lines of code.

Eliminating reference counts Memory management in the MTL library is organized as a reference count system. Although the reference count system is implemented in an efficient way, some overhead is inevitable. When MTL is plugged together with a library with a different memory management schemes—manual allocation or garbage collection—it is therefore desirable to eliminate the extraneous operations that increment and decrement reference counters. Such elimination is at the same time easy to do since all increments and decrements are encapsulated in two member functions, `inc` and `dec`, of a class `refcnt_ptr`. In our experiment, consequently, we simply replaced these two MTL functions by no-op expressions. If the translating compiler removes calls to empty functions, which is fair to assume, the run-time overhead of reference counting thus is completely eliminated and the only remaining overhead is due to the space reserved for the counter variable. In a similar manner we could also have transformed MTL into a garbage-collector compliant library, by additionally replacing all calls to `new` with calls to corresponding allocators. In this experiment two traits are needed to describe the behavior of the functions `dec` and `inc` and two wrappers; the validator comprises 15 lines.

Type-dependent function specialization Suppose a parameterized library is instantiated in such a way that expressions that are meant to contain built-in types now contain user-defined types and have thereby become expensive. Especially with operator expressions the costs of, say, a floating point operator expression and the same expression with a user-defined floating point class differ greatly since

the latter requires the creation, copy, and destruction of a temporary instance of the class type. We conducted a series of experiments where we instantiated MTL with the LiDIA type `bigfloat` and rewrote all expensive operator expressions. The rewrite rules are all similar to the last two in [fig. 1](#), and they are actually applicable not only to `bigfloat` but to most of the numeric types LiDIA supports, including various forms of multiprecision integer, modular, rational, floating point, and complex arithmetic. The LiDIA library provides for each operator of these types a corresponding function that needs no temporaries, and the interfaces of these functions are consistent across the types. We can thus regard these types as belonging to a concept whose requirements include the operator-function correspondences, which allows us to set up concept-based rewrite rules and make them all available for each type with a single declaration that the type belongs to the concept. To give an impression of how much user support was required: the optimization of the LiDIA type `bigfloat` required the completion of 7 traits and the same number of wrappers and approximately 100 lines of code for the validators. These optimizations were applied to core routines in MTL such as a Givens rotation, which resulted in the removal of 12 temporaries and a 6% performance increase for this routine [25].

Non-average best cases Most libraries, including STL, implement sorting using quicksort, because of quicksort’s $\mathcal{O}(N \log N)$ average case complexity with a small constant factor. It is well known, however, that *almost-sorted* sequences can be sorted in linear time by insertion sort, an algorithm one normally avoids because its average complexity is $\mathcal{O}(N^2)$. Almost-sorted sequence are sequences where either only a constant number of elements are out of order or no element is more than k elements away from its proper location, for a constant k . They occur in practice, for example, in cases where insertions and deletions are frequently being made in a large sorted vector without always resorting it, but the vector is resorted after a small number of such updates. If the number of updates is below some constant bound, it is more efficient to resort it with insertion sort than with quicksort. Rather than the application programmer having to complicate his or her code with these considerations, the library designer can provide a set of rewrite rules that accomplish the optimization. In this experiment, two data types are involved with two member functions that depend on each other. Any call of a vector updating function applied to vector `data` is rewritten as the same call augmented with incrementing of a counter, `updates[&data]`, where `updates` is an STL map introduced for the purpose of mapping container addresses to nonnegative integers. For example,

Experiment	Validator	Traits	Wrappers
Bounds checking	10	-	-
Reference counts	15	2	2
Specialization	100 (15)	7	7
Non-avg. best cases	30	1	-

Figure 2. User-provided information: lines of code for rule validations (first column) and number of trait completions and wrappers (second and third column). In the example of function specialization where several rewrite rules are involved the number in parentheses (15) denotes the lines of code on average.

```
data.push_back(v)
```

is rewritten to

```
++updates[&data], data.push_back(v)
```

Calls of the standard library function `sort` (the one implemented with quicksort) such as

```
sort(data.begin(), data.end());
```

are rewritten to

```
frequent_sort(data.begin(), data.end(), data);
```

where `frequent_sort` is defined as

```
template <typename RandomAccessIterator,
          typename Container>
void frequent_sort(
    RandomAccessIterator first,
    RandomAccessIterator last,
    const Container& data)
{
    if (updates[&data] < max_updates)
        insertion_sort(first, last);
    else
        sort(first, last);
    updates[&data] = 0;
}
```

One traits completion is required in this experiment and a validator of 30 lines.

5. Architecture

The *Simplicissimus* framework consists of three parts: the core engine, an interface to the translating compiler, and user-provided information. Beginning with a description of the core engine, we focus in this section on the first two parts and deal with the user input in sect. 7.

The core engine is a stand-alone C++ program of about 19k lines of code that takes an expression and returns its transformation according to a set of (user-defined or built-in) rewrite rules. For the design of the engine it is essential that the input is given in *expression template* format. In short—more detail will be given in the next section—an expression template is a C++ class template that represents an expression in prefix form, annotated with meta-information about its arity, and, most importantly for genericity, with parameterized operands. The following expression template, for example, represents a subscript expression, parameterized by the vector and the index type and annotated with the meta-information `BinaryExpr`:

```
template <typename Vec, typename Index>
class BinaryExpr<Subscript, Vec, Index>
```

Both the actual expression and the expressions on the left and right hand side of a rewrite rule are assumed to be represented in expression template format. Taking the actual expression, the first action of the engine is to perform a syntactic check and to match the input against the left hand sides of the available rules. In this comparison most rules do not match, for example since their left hand side is of different arity than the actual expression or contains a different operator; these rules do not have to be further considered. If a left hand side matches, however, the associated rule potentially can be applied to the input and the engine next initiates a semantic check.

Each rule has a *validator* that specifies the conditions under which the rule can be applied. Hand-crafted for each particular rule, the validator formulates the constraints for the operator, each operand, and the relations between them. It can require, for example, that the types of the operands satisfy certain axioms with respect to the operator or that the operator has no side effects. Most of the specified constraints cannot be derived automatically but depend on the type and function descriptions that users have made available in *traits* template classes (see sect. 6). The semantic check therefore does not consist in directly deriving the constraints but in deriving them as a logical consequence from the descriptions provided; for an efficient derivation the conceptual level of the descriptions and the organization of concepts in hierarchies is crucial. The derivations themselves are entirely transparent to users. In illustration, the following code snippet shows the validator for a rule that rewrites the subscript operator. In this example the validator specifies one constraint, namely that the `Vec` operand behaves like a vector. In general, a validator can express multiple constraints, including user-defined ones, and combine them with logical connectives.

```
struct Validator< Expr<
BinaryExpr<Subscript, Vec, Index> > >
{
    static const bool valid =
        is_vector<typename Vec::result_type>::RET;
};
```

If the actual expression passes the semantic check that is associated with a rule, the application of the rule is guaranteed to be correct. Whether or not the rule is actually applied, however, depends on the underlying rewrite-strategy. Currently implemented are two strategies, `first-fit` and `best-fit`, that apply either the first syntactically and semantically correct rule or the one that, in addition, is the cheapest. In the latter case, users can determine the notion of costs; generally, they can extend the engine by their own strategies. In any case, the engine finally selects a rewrite rule, rewrites the actual expression and returns the result (represented in expression template format). If no rule matches or no rule can be verified, either because the users did not provide any information or because the constraints cannot be shown to be met, the original expression is returned unchanged.

Although expression templates are legal C++ code, source code at the user level typically is not written in expression template format. It is necessary, therefore, to transform normal C++ expressions into equivalent instances of expression templates. Given the complexity of the C++ language, however, handling source code to its full extent is not easy. To avoid the effort of parsing C++ code, a task as tedious as it is non-trivial, we decided to use full-featured compilers and provide interfaces to the compiler's internal representation at the level of the abstract syntax tree. Currently, we have implemented an interface to the Gnu compiler, `gcc`; proprietary restrictions aside, other front ends could be interfaced as well. The Gnu interface consists of two files that provide mappings between two internal representations: from an abstract syntax tree in the internal TREE language of `gcc` [5] to `Simplicissimus`' internal expression templates and, conversely, from expression templates back to TREE expressions. In a relatively straightforward way the interface module recursively walks the tree in one representation and writes out its counterpart in the complementary representation. For the conversion to expression templates, for example, the interface contains the functions

```
build_expr_template_from_variable
build_expr_template_from_literal
build_expr_template_from_tree_node
```

and, conversely, for the simpler task of converting an expression template back, the function

```
build_expr_from_expression_template
```

Fig. 3 depicts how `Simplicissimus` and the Gnu compiler collaborate. The figure also shows that the process of expression transformations is entirely transparent to application users: provided their library designer has set up the required semantic information they simply control `Simplicissimus` with a command line flag:

```
g++ -fsimplify prog.C
```

It remains to explain and justify the choice of expression templates for the internal representation.

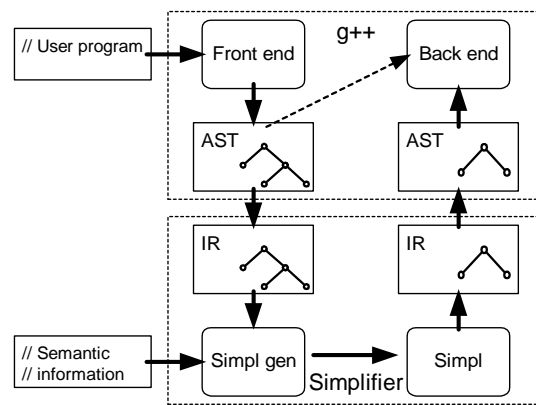


Figure 3. A user expression, translated into Gnu's abstract syntax tree (upper box), mapped to the internal representation in `Simplicissimus`, rewritten by the core engine (lower box), and mapped back.

6. Implementation

It is well known that C++ with its template feature is a two-level language and that the template sub-language is Turing-complete [6]. It is therefore possible to map the transformation problem under consideration from C++ to its subset, the template sub-language, solve it there and map the result back. As we have seen in the last section, this is exactly the approach taken with `Simplicissimus` and ultimately the reason why expression templates are necessary. What, however, is the advantage of switching to the template context? There are in fact two advantages. First, performing library transformations at instantiation time implies that neither the manipulations themselves nor any of the required checks incur any run-time overhead. Second, embedding the core engine in the template language makes its implementation considerably easier. Two of the major parts of the engine, pattern matching and recursive expression walking, are already implemented by the translating compiler to process template instantiation and can therefore simply be reused. Since the techniques applied in `Simplicissimus` are known as *template metaprogramming*, which has become a well-established technique in the theory and practice of C++ libraries, we only summarize the two most important idioms used—the already mentioned expression templates and traits—and refer the interested reader to the literature and programming examples elsewhere [6, 8].

Parameterization in C++ includes the possibility of partial specialization that both expression templates and traits essentially depend on. From the language specification:

A primary class template declaration is one in which the class template name is an identifier. A

template declaration in which the class template name is a template-id, is a partial specialization of the class template named in the template-id. A partial specialization of a class template provides an alternative definition of the template that is used instead of the primary definition when the arguments in a specialization match those given in the partial specialization ([1], 14.5.4)

The original motivation for partial specialization was to allow users to express different levels of genericity. Given several template definitions, the language processor then ensures that the most specialized definition is selected for instantiation. Another way to look at partial specialization is as a way to realize branching. A two-way decision for example can be simulated using two template definitions, one in terms of a universally quantified template type parameter, and another, more specialized one; the more general definition thereby represents the default behavior and the specialized definition the alternative behavior.

Returning to *Simplicissimus*, partial specialization implies that it suffices just to *list* all rewrite rules. Provided they are phrased as specializations of the same class template, the template processor then automatically selects the one whose left hand side fits best. For such exploitation of the translating compiler to work, however, an expression at user's level has to become a template parameter.

Values, function identifiers, and operators are not allowed as template parameters. How, then, is it possible to transmute an expression to a template parameter? The idea is to represent expressions symbolically. If its variables are represented symbolically, with different symbols for different variables, and its function or operator identifier by a statically resolvable tag, then the template processor can evaluate them and the expression can serve as a template parameter. Expression templates, now, model this very idea of expressions as types. They represent

- variables as classes, `Var<0>`, `Var<1>`, ..., with different classes for different variables; literals follow the same scheme;
- function identifiers or operators as *tags*, that are empty classes with mnemonic identifiers.

Additionally, as we have seen, expression templates use prefix notation and wrap up an expression in a class that indicates its arity (`UnaryExpr`, `BinaryExpr`, ...). It should be clear that the definitions can be recursively continued to a nested set of templates and that the process of template instantiation creates objects that represent parse trees of arbitrary depth.

Historically, expression templates were introduced in the context of numeric computations, to eliminate temporaries

especially in operator expressions [30], and were first implemented in high-performance libraries [29, 24]. Subsequently, programmers understood that the template processor essentially works as an interpreter of the template sub-language and started borrowing ideas from symbolic computation. Although the purpose of expression templates in *Simplicissimus* is different from the original one it shares with it the idea of compile-time computations. Of further importance for us is that the encoding of an expression as expression template is reversible and allows for translations back to "normal" C++ expressions.

Now, if the template processor essentially takes over the syntactic check of a rewrite rule, can it also support the semantic check? At first glance, the chosen expression template format and its inherent genericity provides a problem. If a rule is written for an expression that can be instantiated with operands of different types, with types that are not even known yet, how should the semantic check be expressed? Which type name should it refer to and what members are available for this type? Interface templates, *traits*, come to the rescue here. They allow the core engine to express the semantic check independently from any particular type and to abstractly lay out the various steps of the semantic check.

The traits technique "provides a convenient way to associate related types, values, and functions with a template parameter type without requiring that they be defined as members of the type" [22]. Traits are characterized by two properties. First, they are based on partial specialization, and second, they are families of class templates that all have the same interface, e.g., common type definitions or static data members. With these two properties combined, traits can provide uniform interfaces on the one hand and encapsulate specific behavior on the other hand. The following trait is an illustration of one of the traits used in *Simplicissimus* to encapsulate the computational behavior of operators; among others its members include `is_applicative` and `can_underflow`. This trait is specialized with many operators, each of which declares whether or not it is applicative or can underflow. For the operators that do not have a partial specialization defined, the primary template applies, which implements a default behavior and sets, for example, the two members above, `is_applicative` and `can_underflow`, to true by default. In the following example we list the partial specialization of the subscript member function of the vector class:

```
class BinaryOpTraits<Subscript> {
public:
    typedef __true_type    is_applicative;
    typedef __false_type  has_side_effects;
    typedef __true_type    lhs_is_const;
    typedef __true_type    rhs_is_const;
    typedef __false_type  can_overflow;
    typedef __false_type  can_underflow;
```

```
typedef __false_type can_zero_divide;
};
```

As we pointed out already, most parts of a type or function description in *Simplicissimus* cannot be automatically derived—while the right hand side of a type definition is given through a traits class, its left hand side is type- or function-dependent and users are responsible for its value. Technically, on the other hand, it is very easy to provide a traits definition since it comes down to filling out a template.

Traits were originally used to overcome problems with the internationalization of C++. They are now part of the language specification of I/O classes, strings, floating points, characters, and several other built-in and library types [1]; they have become one of most widely used techniques in template programming. In *Simplicissimus*, where the semantic check is implemented in terms of interfaces, they allow for expressing the check at an abstract level and for reducing computing costs: only at instantiation time does the particular value of the partial specialization have to be retrieved.

7. User interface

Now, exactly how much do users have to do? To show what steps are necessary to extend *Simplicissimus* by a new rewrite rule we now discuss a complete example; for the code listing see appendix A. We choose an example we already discussed, the elimination of reference counts of MTL. Let us assume that we already know that we want to replace the two member functions `inc` and `dec` by no-op functions and start over from there. The task ahead breaks down into

- defining the rewrite rule;
- filling out two operator traits that describe the functions involved;
- providing wrappers for each function.

Up front we introduce two tags

```
struct IncRefCount {};
struct DecRefCount {};
```

the mere purpose of which is to represent the two functions `inc` and `dec` in all contexts where expressions are expected to be types.

7.1. The rule definition

Rules are implemented as classes that hold the right hand side of a rewrite rule and its validator; the left hand side of the rule is part of the validator. Let us therefore set up a class

`RemoveRefCount` and begin with the definition of its validator. Since we, strictly speaking, specify two rules at the same time—for the increment operator and the decrement operator—we have to design validators for each of them. The validators themselves, however, are easy to define. We only have to make sure that we are actually dealing with the intended function. Here is the validator for the increment operator:

```
template<typename Operand>
struct Validator< Expr<UnaryExpr<
    IncRefCount, Operand> > >
{
    static const bool valid = true;
};
```

The corresponding validator for the decrement operator looks similar; as the reader will notice, both are member classes of the class `RemoveRefCount`. All expressions different from increment and decrement expressions we catch with another member class, a default template that guarantees that the semantic check fails for them:

```
template<typename ExprT>
struct Validator
{
    static const bool valid = false;
};
```

The validator is now defined. The second part of the definition of the `RemoveRefCount` class requires the specification of the right hand side of the rewrite rule, a no-op expression in our case. This again is easy to do since empty expressions are already predefined in *Simplicissimus*. We grab the expression template type that represents the empty expression and declare it to be the result of the rewrite rule. As in the case of the validators, the corresponding type definition, `result`, is encapsulated in a member class:

```
template<typename>
struct Result
{
    typedef Expr<GeneratorExpr<NullOp> >
        result;
};
```

The definition of the class `RemoveRefCount` is now completed. What is left to do is to add the new rule to the set of existing ones. The following lines suffice:

```
#define NEW_RULE RemoveRefCount
#include <simplifier/add_simp_rule.h>
#undef NEW_RULE
```

7.2. Operator descriptions

With the last lines at the end of the previous section the new rule has become available in the system. The increment and decrement operators themselves, however, are not

known to the system until we have provided descriptions of their behavior. For that purpose we specialize one of primary templates that the core engine prepares for operators, the `UnaryOpTraits` in our case. The `UnaryOpTraits` class includes seven type definitions:

- the result type;
- information about the possible change in the control flow caused by exceptions (`can_zero_divide`, `can_underflow`, `can_overflow`);
- information about the possible change of state (`operand_is_constant`, `is_applicative`, `has_side_effects`).

It also includes a function `apply`, internally used during the conversion from expression templates back to “normal” expressions that invokes the original operator. The type definitions allow writing validators that require protection against certain exceptions or state changes. Although the validators in our example are simple and the core engine in fact only needs the member `apply` we will provide a complete definition of the trait class. As the following code shows, we negate that the increment operator is applicative, state that it has side effects, and assert that it does not throw any of the exceptions; in the body of `apply` we store the original function invocation:

```
template<typename Object>
struct UnaryOpTraits<IncRefCount,
                    refcnt_ptr<Object>>
{
    typedef __false_type is_applicative;
    typedef __true_type  has_side_effects;
    typedef __false_type operand_is_const;
    typedef __false_type can_overflow;
    typedef __false_type can_underflow;
    typedef __false_type can_zero_divide;
    typedef void result_type;

    inline static void
    apply(refcnt_ptr<Object>& operand)
    {
        operand.inc();
    }
};
```

The trait definition for the decrement operator looks similar. Of course, if someone else had already provided these traits, for example in the context of the definition of another rewrite rule, we would not have to define them again. In any case, as soon as the rewrite rule and all operators involved are specified, the core engine has all information it needs. The last required step, definition of wrappers, therefore is required not so much by `Simplicissimus` itself but by its integration into the Gnu compiler.

7.3. Wrappers

At the very least we have to provide wrappers for each operator. The body of a wrapper is very simple: it generates the parse tree of the expression template that the respective operator represents. In case of the increment operator, for example, it returns an instance of a unary expression template, specialized by the increment operator:

```
return Expr<UnaryExpr<IncRefCount,Operand>>();
```

In this example, the `Operand` parameter of the expression template is unbound. We know, on the other hand, that in every actual expression template it has to be bound to an instance of the `refcnt_ptr` class, since the increment operator is a member of this class. To ensure that the semantics of the original expression is preserved, the core engine provides a class `TypeCheck`. The wrapper’s task is to take the actual operand type and to pass it on to this type-checking class, along with the formal operand type:

```
TypeCheck<Operand,refcnt_ptr<Object> >
```

To communicate with the translating compiler, finally, the name of the wrapper has to follow the name convention that is used for the internal name mangling scheme:

- member functions are prefixed with `__memfn__`;
- free functions are prefixed with `__exprt__`

Putting together the name mangling scheme, the initiation of the type check, and the correct C++ namespace qualification, the interface of the wrapper for the increment (and likewise for the decrement) operator reads as follows:

```
__memfn__inc(TypeCheck<Operand,
              refcnt_ptr<Object> >)
```

8. Related work

Much work is going on in applying ideas from program transformation to large software systems, especially component-based ones. From a more theoretical point, the approach in `Simplicissimus` borrows from ideas in program specialization and partial evaluation [12], while from a software-engineering view it bears similarities to modern software methodologies, most notably intentional programming, subject-oriented programming, adaptive and aspect-oriented programming [15, 16, 18, 23, 27]. Probably closest to our approach is aspect-oriented programming (AOP), insofar it advocates a software architecture where the “cross-cutting aspects” are cleanly modularized and a “weaver” combines aspects and components to executable code. In fact, our core engine can be viewed as a weaver. The major conceptual difference comes from our application to libraries or, more generally, from the situation where software designer and software user are temporally separated.

Misusing the client-server terminology, one could say that we rewrite code on the client side, while AOP rewrites it on the server side. For non-parameterized data types this difference does not matter; for parameterized types, however, the particular rewrite typically depends on the particular instantiation of a type, thus cannot be dealt with on the server side.

Another closely related area is the field of program transformation by rewriting, especially at the level of the control-flow graph of a program. Graphs form the intermediate representation for example of the optimizer generator OPTIMIX, which specifies and performs program optimizations as graph transformations [2]; also see [3].

Graph rewriting is on the one hand more difficult than the rewriting of expression trees that we perform and is capable of reasoning over data-flow properties [17], which is difficult to do with programs represented in expression tree format. To our knowledge, however, current graph rewrite systems are restricted to predefined types and therefore operate at a lower level of abstraction than our approach.

Extensibility, at the same time, is a trend in compiler construction. The Broadway compiler, for example, can be controlled by an annotation language for optimizing libraries [10]. Closer to a mainstream language but limited to one type, the ROSE C++ source-to-source preprocessor [7] is extensible by a small generator for optimization specifications of class arrays. Finally, the system Magik [9] performs code optimization but can also incorporate into the compiler the meaning of library interfaces and data structures, thus allowing them to be checked for errors to the same degree as for built-in constructs. A key difference of our approach lies in our organization of optimizing transformations in terms of concepts, which allows a library designer to augment a library with many transformations and enable them for any new type the library defines merely by identifying the most general concept to which the type belongs.

9. Availability and Acknowledgment

The source code and more information are available at the Simplificissimus web page <http://www.cs.rpi.edu/research/gpg/Simplificissimus>. This work is sponsored in part by SGI, Mountain View, California.

References

- [1] I. F. 14882. *International Standard for the C++ Programming Language*. American National Standards Institute (ANSI), X3 Secretariat, 1250 Eye Street NW, Suite 200, Washington, DC 20005, 1998.
- [2] U. Abmann. *OPTIMIX, A Tool for Rewriting and Optimizing Programs*. Chapman-Hall, 1998.
- [3] U. Abmann and A. Ludwig. Aspect Weaving by Graph Rewriting. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, number 1799 in Lecture Notes in Computer Science, Erfurt, Oct. 1999.
- [4] M. Austern. *Generic Programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1999.
- [5] CodeSourcery, LLC. *G++ Internal Representation*, August 2000. <http://gcc.gnu.org/onlinedocs>.
- [6] K. Czarnecki and U. W. Eisenecker. *Generative Programming—Towards a New Paradigm of Software Engineering*. Addison Wesley Longman, 2000.
- [7] K. Davis and D. Quinlan. ROSE II: An optimizing code transformer for C++ object-oriented array class libraries. In *Workshop on Parallel Object-Oriented Scientific Computing (POOSC'98), at 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1543 of LNCS. Springer Verlag, 1998.
- [8] B. Dawes and D. Abrahams. Boost. <http://www.boost.org>.
- [9] D. R. Engler. Interface compilation: steps toward compiling programs. *Transactions on Software Engineering*, 25(3), 1999.
- [10] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In T. Ball, editor, *2nd Conference on Domain-Specific Languages*. Usenix, 1999.
- [11] S. G. Inc. Standard Template Library Programmer's Guide. <http://www.sgi.com/tech/stl/>, 1997.
- [12] N. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.
- [13] D. Kapur and D. R. Musser. Tecton: A language for specifying generic system components. Technical Report 92-20, Rensselaer Polytechnic Institute Computer Science Department, July 1992.
- [14] D. Kapur, D. R. Musser, and A. A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proceedings of a Workshop on Program Specification*, volume 134 of LNCS, pages 402–414, Aarhus, Denmark, Aug. 1981. Springer.
- [15] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP'01)*, 2001.
- [16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwi. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.
- [17] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *International Conference on Compiler Construction*, Springer Lecture Notes in Computer Science, pages 52–68, 2001.
- [18] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.
- [19] S. B. Lippman. *C++ Gems*. Cambridge University Press, December 1996.
- [20] D. Musser and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

- [21] D. Musser, S. Schupp, and R. Loos. Requirements-oriented programming. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming—International Seminar, Dagstuhl Castle, Germany 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2000.
- [22] N. Myers. A new and useful template technique. In *C++ Gems* [19].
- [23] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proc. of the 1992 Conf. on Object-oriented Programming Systems, Languages and Applications*, 1992.
- [24] J. V. Reynnders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA: A framework for scientific simulations on parallel architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.
- [25] S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. User-extensible simplification—type-based optimizer generators. In R. Wilhelm, editor, *International Conference on Compiler Construction*, Springer Lecture Notes in Computer Science, pages 86–101, 2001.
- [26] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.
- [27] C. Simonyi. The future is intentional. *IEEE Computer*, 1999.
- [28] The LiDIA Group. Lidia—a C++ library for computational number theory. <http://www.informatik.tu-darmstadt.de/TI/LiDIA/>.
- [29] T. Veldhuizen. Blitz++. <http://oonumerics.org/blitz>.
- [30] T. Veldhuizen. Expression templates. In *C++ Gems* [19].
- [31] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, Dordrecht-Boston, 1982.
- [32] R. Wille. Concept lattices and conceptual knowledge systems. *Computers and Mathematics with Applications*, 23:493–522, 1992.

A. Reference count elimination

```
#ifndef MTL_REMOVE_REFCNT_HEADER
#define MTL_REMOVE_REFCNT_HEADER

#include <simplifier/simplify.h>
#include <mtl/refcnt_ptr.h>

namespace expr_info {
    namespace mtl {
        struct IncRefCount {};
        struct DecRefCount {};
    }

    using namespace ::expr_info::mtl;
    using namespace ::mtl;
```

```
template<typename Object>
struct UnaryOpTraits< DecRefCount,
    refcnt_ptr<Object> > {
    typedef __false_type is_applicative;
    typedef __true_type has_side_effects;
    typedef __false_type operand_is_const;
    typedef __false_type can_overflow;
    typedef __false_type can_underflow;
    typedef __false_type can_zero_divide;
    typedef void result_type;

    inline static void
    apply(refcnt_ptr<Object>& operand)
        { operand.dec(); }
};

template<typename Object>
struct UnaryOpTraits< IncRefCount,
    refcnt_ptr<Object> > {
    typedef __false_type is_applicative;
    typedef __true_type has_side_effects;
    typedef __false_type operand_is_const;
    typedef __false_type can_overflow;
    typedef __false_type can_underflow;
    typedef __false_type can_zero_divide;
    typedef void result_type;

    inline static void
    apply(refcnt_ptr<Object>& operand)
        { operand.inc(); }
};

}

namespace simplifier {
    namespace mtl {
        using namespace expr_info;

        struct RemoveRefCount {
            typedef StandardRuleClass rule_class;

            template<typename ExprT>
            struct Validator {
                static const bool valid = false;
            };

            template<typename Operand>
            struct Validator<Expr<UnaryExpr<
                IncRefCount, Operand> > > {
                static const bool valid = true;
            };

            template<typename Operand>
            struct Validator<Expr<UnaryExpr<
                DecRefCount, Operand> > > {
                static const bool valid = true;
            };

            template<typename>
            struct Result {
                typedef Expr<GeneratorExpr<NullOp> >
```

```

        result;
    };
};
}

#define NEW_RULE RemoveRefCount
#include <simplifier/add_simp_rule.h>
#undef NEW_RULE
}

namespace mtl {
    template<class Operand, class Object>
    inline Expr<UnaryExpr<IncRefCount,
        Operand> >
    __memfn__inc(TypeCheck<Operand,
        refcnt_ptr<Object> >)
    { return Expr<UnaryExpr<IncRefCount,
        Operand> >(); }

    template<class Operand, class Object>
    inline Expr<UnaryExpr<DecRefCount,
        Operand> >
    __memfn__dec(TypeCheck<Operand,
        refcnt_ptr<Object> >)
    { return Expr<UnaryExpr<DecRefCount,
        Operand> >(); }
}
#endif

```