Scientific Programming 11 (2003) 309–320
IOS Press

Design patterns for library optimization

Douglas Gregor[a], Sibylle Schupp[b,1] and David R. Musser[a]
[a]*Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180, USA*
E-mail: {gregod,musser}@cs.rpi.edu
[b]*Department of Computing Science, Chalmers University of Technology, SE-41296 Gothenburg, Sweden*
E-mail: schupp@cs.chalmers.se

Abstract. We apply the notion of design patterns to optimizations performed by designers of software libraries, focusing especially on object-oriented numerical libraries. We formalize three design patterns that we have abstracted from many existing libraries and discuss the role of these formalizations as a tool for guiding compiler optimizers. These optimizers operate at a very high level that would otherwise be left unoptimized by traditional optimizers. Finally, we discuss the implementation of a design pattern-based compiler optimizer for C++ abstract data types.

1. Introduction

Design patterns have been widely accepted as an invaluable tool for the design of software systems. They represent abstract notions of the behavior of code without collapsing under the weight of implementation details, and therefore serve as an efficient method of communicating design. Design patterns are not synthesized but instead are abstracted from commonalities in design found amongst many successful software systems. As abstractions, these design patterns must be customized for any specific task at hand, but any instance retains the properties of the design pattern(s) applied.

Design patterns need not be limited to high-level design. Techniques employed by designers of high-performance software libraries to enable code optimizations also constitute design patterns. Especially in object-oriented libraries, there are standard ways for example to minimize the number of temporaries, to manipulate the evaluation of an expression, or to choose among functionally equivalent expressions. It is essentially because of these optimization patterns that libraries in higher level programming languages such as C++ or Java have become competitive with those written in C or Fortran. Often, however, the price for using these patterns is compromised code clarity.

In an object-oriented numeric library, for example, it is often possible to directly express mathematical formulae by using operators on user-defined types, but these operator expressions are known to cause a large number of extraneous temporary values to be computed and stored. While these temporaries may be inexpensive for fundamental integer or floating-point types, or even small user-defined types, such as complex numbers, temporaries for large user-defined types, such as arbitrary-length integers, arbitrary-precision floating point numbers, or matrices, can become very costly. Programmers have reacted to these extra costs by reverting from the more natural operator-centric representation of mathematical expressions to the use of procedure calls that require fewer temporaries and result in better overall performance.

Design patterns for optimization provide a new perspective on the ways in which library authors design code for maximal performance. These optimization patterns offer the same benefits as traditional design patterns in that they succinctly communicate design, but have additional value in that they can be directly transformed into optimization opportunities for compilers. They are based on the observation that the transformation of, e.g., an operator-centric expression to an equivalent procedural form is a largely mechanical task for the programmer, which, however, cannot be automated as long as the programmer cannot communicate to the compiler the kind of transformation it should per-

[1]This work was performed while the second author was at Rensselaer.

ISSN 1058-9244/03/$8.00

form. What is needed for automation is an optimization scheme a programmer can refer to and a categorization of related optimizations, including the semantic conditions under which they can be applied.

Optimization patterns help make the process of specifying such transformations manageable by defining an abstract form that these transformations may be derived from. Assuming a compiler supports a particular optimization pattern, a user (i.e., library designer) can refer to this pattern and identify the characteristics that make a given transformation an instance of this design pattern. Conversely, an optimization that is given in the form of an optimization pattern has been proved to be applicable across several libraries, and thus has established itself as an optimization methodology. It is therefore worthwhile to develop compiler optimizers based on design patterns. Our Simplicissimus project [19] has already produced one such compiler optimizer that can handle optimization patterns; we hope that other open compilation environments will follow.

We have surveyed several C++ object-oriented numeric libraries and abstracted design patterns that are common amongst these libraries. In this paper we introduce three patterns that are important, but not restricted to numerical applications: the Replacement pattern, the Assignment Replacement pattern, and the Temporary Removal pattern. The Replacement pattern is a general pattern for rewriting expressions as other expressions; the latter two patterns can be understood as refinements of the Replacement pattern that provide additional optimization opportunities.

We begin the presentation with examples of optimizing designs gathered from C++ object-oriented numeric libraries in Section 2 that motivate the abstraction that underlies each pattern. In Sections 3 and 4 we formalize, discuss, and illustrate the Replacement pattern and two subpatterns called Assignment Replacement and Temporary Removal. Section 5, finally, summarizes the implementation of optimization patterns within the Simplicissimus framework and briefly describes its integration into the GNU C++ compiler. The emphasis of the paper, however, is on the concept of a design pattern for optimization, and the main purpose of the paper is to initiate the identification and refinement of these patterns.

2. Optimization methods used by library designers

We surveyed several object-oriented numeric libraries, including LiDIA [20], the Matrix Template Library (MTL) [17,18], the Number Theory Library (NTL) [16], and the Basic Linear Algebra Subprograms (BLAS) [12], searching for design patterns commonly used to facilitate optimizations that could be leveraged by a compiler optimizer instead of relying on the library user. The most common technique is the use of procedures or functions in lieu of operator expressions. These functions can be placed into roughly three categories: shorthand functions, operations that write their result directly to a target, and functions that combine several operations into one call. Each of these categories will be further described with examples from the aforementioned libraries.

Library	Operation	Semantics
NTL	Inverse(x)	$1/x$
LiDIA	x.AssignZero()	$x := 0$
LiDIA	x.EqualsOne()	$x = 1$

Fig. 1. Shorthand operations.

Throughout this paper, by *semantic equivalence* of two expressions we mean equivalence of the observable behavior of the expressions. Expressions e_1 and e_2 have the same observable behavior if replacing an instance of one with the corresponding instance of the other will not change a program barring exceptional conditions (e.g., memory allocation failure). We denote this relation by $e_1 \equiv e_2$.

In addition to standard mathematical notation, we use the infix copy assignment operator ':=' that replaces the value of the left-hand operand with the result of computing the right-hand operand. The result of this operation is the left-hand operand.

2.1. Shorthand functions

Shorthand functions often encapsulate operations that are expressible by common operations but may be computed more efficiently within a single function. Such operations include complex conjugation, inverses, and taking the square of a value. Figure 1 illustrates examples of shorthand functions in NTL and LiDIA.

2.2. Targeted operations

The return value of an operation is often the cause of unwanted temporaries. Even in simple assignments, such as $y := a \times x$, a temporary is generated by the multiplication $a \times x$ and must be copied into y. As a reaction to this, library authors create procedures that store the result directly into one of its operands. Figure 2 illustrates some examples of this pattern.

Library	Operation	Semantics
LiDIA	`add(x, y, z)`	$x := y + z$
LiDIA	`multiply(x, y, z)`	$x := y \times z$
NTL	`Sub(x, y, z)`	$x := y - z$
NTL	`Inverse(x, y)`	$x := 1/y$
MTL	`transpose(A, B)`	$B := A^T$

Fig. 2. Targeted operations.

2.3. Composite operations

Certain sets of operations are often used in conjunction. Library authors have used this as an opportunity to introduce new functions that perform all operations in one step without the creation of temporaries and with efficiency that would otherwise not be achieved using separate functions. The most obvious implementation of this technique is in the BLAS libraries, where operator expressions are not included but instead complicated general-purpose routines are supplied. Some examples of composite functions are listed in Fig. 3.

3. The replacement pattern

Shorthand, targeted, and composite operations often have semantics that are expressed via mathematical formulas. In the majority of object-oriented numeric libraries, these mathematical formulas are also directly expressible, but come at a cost in efficiency. The programmer is expected to transform the mathematical formulas into a set of function or procedure calls to evaluate them. Informally speaking, the Replacement pattern is a natural abstraction of this expression transformation for optimization and can be likened directly to a rewriting system where the left-hand side of a rewrite rule denotes the mathematical expression and the right-hand side denotes the equivalent, more efficient, procedure call. In the rest of this section we formalize the Replacement pattern in terms of sets of rewrite rules.

3.1. Definitions and notation

Expressions are finite tree structures built from a given finite set F of function symbols and a denumerably infinite set V of variable symbols; the set of all such expressions is denoted $T(F, V)$. An equation is a pair of such expressions, say (t_1, u_1), usually written $t_1 = u_1$, and the equality rules of inference are captured in the notion of *rewriting* a subexpression of an expression using an equation as a *rewrite rule*. Specifically, a pair of expressions (l, r) is a rewrite rule if l is not just a variable and the variables that appear in r also appear in l. We usually write the rule as $l \to r$, and l is called the left-hand side and r the right-hand side of the rule. Note that in some cases an equation $t = u$ could be used as a rewrite rule as either $t \to u$ or $u \to t$.

A *substitution* is a mapping σ from expressions to expressions that is determined entirely by its value on a finite number of variables; a substitution is denoted by an expression of the form $\{t_1/v_1, \ldots, t_k/v_k\}$, read "substitute t_1 for $v_1, \ldots, t_k$ for v_k." The $k \geqslant 0$ variable symbols $v_1, \ldots, v_k$ must be distinct, and the case $k = 0$ is the identity substitution ι such that $\iota(t) = t$ for all expressions t. Following convention we write an application of a substitution as $t\sigma$ rather than $\sigma(t)$. For example, if $\sigma = \{(a + b)/x, (b \cdot c)/y, d/z\}$ and $t = x+(a\cdot(z-y))$, then $t\sigma = (a+b)+(a\cdot(d-(b\cdot c)))$.

To define rewriting precisely we also need some notion of position of an occurrence of a subexpression s within an expression t. A standard way [4] to define a position p in an expression t is as a sequence of natural numbers: the root position is assigned the empty sequence, denoted by Λ; and if p is the position in t of a subexpression $s = f(s_1, \ldots, s_k)$ then the subexpression s_i in the ith argument of s is assigned position $p.i$ in t, for $i = 1, \ldots, k$. Then $t[p]$ denotes the subterm of t at position p, and $t[p \leftarrow s]$ denotes the result of replacing the subexpression at p in t by the expression s. For example, if $t = x + (a \cdot (z - y))$ then $t[2.2.1] = z$, and $t[2.2.1 \leftarrow d] = x + (a \cdot (d - y))$.

For a given rewrite rule $l \to r$, a relation on pairs of expressions, *t rewrites to u*, can be defined as: for some position p in t, there is a substitution σ such that $t[p] = l\sigma$ and $u = t[p \leftarrow r\sigma]$. We write this as "$t \to u$ using $l \to r$," overloading the use of the symbol $\to$. For example,

$$x + (a \cdot (z - y)) \to x + ((z - y) \cdot a))$$

using $i \cdot j \to j \cdot i$, since, for $\sigma = \{a/i, (z - y)/j\}$,

$$(x + (a \cdot (z - y)))[2] = (a \cdot (z - y)) = (i \cdot j)\sigma$$

and

$$x + ((z - y) \cdot a)) = (x + (a \cdot (z - y)))[2 \leftarrow (j \cdot i)\sigma].$$

For a given set of rewrite rules $\mathcal{R}$, we say $t \to u$ using $\mathcal{R}$ if $t \to u$ using $l \to r$ for some rule $l \to r$ in $\mathcal{R}$. These definitions can be extended to *conditional rewriting*: a conditional rewrite rule is a triple of expressions (l, r, c) where (l, r) is a rewrite rule and c is a predicate expression whose variables also appear in

Library	Operation	Semantics
MTL	`mult(A, x, y, z)`	$z := A \times x + y$
MTL	`mult(A, B, C)`	$C := A \times B + C$
BLAS	`AXPY(a, x, y)`	$y := a \times x + y$
BLAS	`GEMM(a, A, B, b, C)`	$C := a \times A' \times B' + b \times C'$ where $x' = x, x^T$, or x^H

Fig. 3. Composite operations. The BLAS library's GEMM subroutine has been simplified from its original thirteen arguments for brevity.

l. We usually write the rule as $l \to r$ (if c). For a set of such conditional rules $\mathcal{R}$ the rewriting relation $t \to u$ using $\mathcal{R}$ is defined by $t \to u$ if there is a rule $l \to r$ (if c) in $\mathcal{R}$ such that $t \to u$ using $l \to r$ and $c\sigma$ is true, where σ is the same substitution used in the rewrite.

The nature of the condition on a rewrite rule depends partly on the programming language used and its type system, partly on the program transformation in which the expression e takes place. Conditions can include conceptual or type requirements as well as the specification of computational behavior, e.g., freedom from side-effects of a functional expression, or anti-aliasing of pairs of variables. We want to emphasize, however, that especially in the examples listed the validity of a condition cannot (efficiently) be deduced in an automated way. What can be automatically checked, however, are assertions of properties, including the logical implications of these assertions. We therefore assume that the pattern designer asserts certain properties of variables and other subexpressions, and that a condition is then checked against these declarations. Likewise it is the pattern designer, and not a program, that claims the semantic equivalence of two expressions.

3.2. The replacement pattern

We assume there is a *cost* function available from expressions to reals (or any totally ordered domain) so that costs of expressions can be compared. We also recall the relation of semantic equality, $\equiv$, as introduced in Section 1.

Definition. *Let L and R be expressions and P be a predicate expression. A Replacement pattern is a triple*

$$(L, R, P)$$

such that

1. $L \equiv R$ *whenever* P *holds*
2. *cost*$(R) \leqslant$ *cost*(L)

Operationally speaking, a Replacement pattern can be implemented in a framework of conditional rewrite rules. Some patterns can be implemented as a single conditional rewrite rule, $L \to R$ (if P). This is the case, for example, with the shorthand operation Inverse in Fig. 1, with the rewrite rule

$$1/x \to \text{Inverse}(x)$$

where there is no condition required. Similarly, the Replacement patterns for the other two shorthand operations in Fig. 1 can each be implemented with a single rule. More generally, the implementation of a Replacement pattern can require several rules if there are expressions that are semantically equivalent to L that are not instances of L in the strict syntactic sense of matching defined by the rewrite system. Consider, for example, the Replacement pattern instance that targets the BLAS AXPY routine in Fig. 3, which is commonly used for manipulation of vectors. Formally, this instance is

$$(y := a \times x + y,\ AXPY(a, x, y), P(a, x, y))$$

(To simplify the discussion in this section we do not spell out the constraints represented by the predicate P; details of such constraints in several examples are however discussed in Section 5.) We can use this triple first of all to form the rewrite rule

$$(y := a \times x + y) \to AXPY(a, x, y)$$
$$(\text{if } P(a, x, y)),$$

but if we want the same optimization in the case $a = 1$ we also need the rule

$$(y := x + y) \to AXPY(1, x, y)\ \ (\text{if } P(1, x, y)),$$

since $y := x + y$ doesn't syntactically match $y := a \times x + y$ (because it lacks an occurrence of the multiplication operator, $\times$). Similarly, to reflect the role of commutativity of $+$ in semantic equivalence of expressions, we need two more rules

$$(y := y + a \times x) \to AXPY(a, x, y)$$
$$(\text{if } P(a, x, y)),$$
$$(y := y + x) \to AXPY(1, x, y)$$
$$(\text{if } P(1, x, y)).$$

(We could get by without such additional rules if we were implementing in terms of a more powerful form of rewriting, such as associative-commutative rewriting [4].)

Thus, in general, to implement the Replacement pattern (L, R, P) we require a set of n rewrite rules $l_i \rightarrow r_i$ (if c_i) such that $l_i \equiv r_i \equiv L\sigma_i$ and $c_i \equiv P\sigma_i$ for some substitution σ_i, for $i = 1, \ldots, n$.

Given that pattern designers are responsible for determining the semantic equality of left- and right-hand sides as well as for identifying the constraints that hold for instances of L, the rewrite framework is left with three tasks. First, it performs the syntactic match between an actual subexpression s and a left-hand side, l, of one of the rewrite rules $l \rightarrow r$ (if c), obtaining a substitution σ such that $l\sigma = s$. Second, it checks the constraints $c\sigma$ by inferring whether or not declaratively asserted properties of the actual subexpression s preserve the constraints. If the constraints are satisfied, it applies the rewrite rule and replaces s with the corresponding instance $r\sigma$.

Note that for a set of rewrite rules and a given actual expression the selection of an appropriate rewrite rule is not necessarily unique: the actual expression can match, and satisfy the conditions, of more than one left-hand side. The cost function associated with each rule can then be used to compute an optimal selection.

All examples we have seen so far can be considered as instances of the Replacement pattern. This is unsurprising, as the Replacement pattern itself relies on the very general notion of expression rewriting, but it nonetheless serves as a base for pattern-based optimizers. Several of these examples, however, share additional characteristics and furthermore refer to expression schemes that occur sufficiently frequently to establish patterns on their own, or, more precisely, *subpatterns* of the Replacement pattern. A subpattern inherits all properties of its superpatterns but may add properties, both to the left-hand side of its superpattern and to its right-hand side. Any optimization for a given pattern is valid for any subpatterns of that pattern, and the subpattern relationship is necessarily transitive. The next section presents two examples of subpatterns.

4. Assignment replacement and temporary removal

As the survey in Section 2 has shown, a great deal of emphasis within numerical computing is placed on the removal of temporaries. Therefore, many instances of the Replacement pattern within numeric libraries are designed specifically to remove extraneous temporaries. In this section we first introduce the Assignment Replacement pattern, an abstraction from the targeted operation discussed earlier, then the Temporary Removal adaptor that further eliminates temporaries by adding appropriate expressions. While the Assignment Replacement pattern is a syntactic refinement of the Replacement Pattern, the Temporary Removal adaptor applies to Replacement patterns and generates new instances of Assignment Replacement patterns (which are then eligible for optimization with existing pattern instances).

4.1. The assignment replacement pattern

As motivation we again consider the Replacement pattern for the BLAS routine, $(y := ax + y, AXPY(a, x, y), P)$. We consider here just the first of the four rewrite rules that implement this pattern as discussed in the previous section.

$$(y := a \times x + y) \rightarrow AXPY(a, x, y) \ \ (\text{if } P).$$

If we consider the naive computation of the expression $y := a \times x + y$, three loops are required for evaluation: one for the scalar multiplication, one for the vector addition, and one for the vector copy. For each of the two temporaries created by this expression, memory for the vector's storage must be allocated and later freed by the destruction of the temporary. On the other hand, the procedure call $AXPY(a, x, y)$ requires no temporaries and a single loop. Since the discussion of targeted expressions in Section 2.2 has shown that copy assignments are a frequent source of temporaries (see Fig. 2) the introduction of a separate optimization pattern for copy assignments seems to be appropriate.

Definition. *An Assignment Replacement pattern is a Replacement pattern* (L, R, P) *such that the root of* L *is a binary function* (*operator*) *that represents an assignment to its left operand.*

As with the Replacement pattern, instances of the Assignment Replacement pattern may vary greatly in generality and scope. The LiDIA routine `add` may only be useful for the expression listed in Fig. 2, whereas the BLAS routine GEMM has many possible instances, as is illustrated in the form of rewrite rules in Fig. 4.

The utility of the Assignment Replacement pattern is not in its ability to express optimizations not available to the Replacement pattern, but to simplify the

$(C := a \times A \times B + b \times C)$	$\rightarrow GEMM(a, A, B, b, C)$
$(C := A \times B + b \times C)$	$\rightarrow GEMM(1, A, B, b, C)$
$(C := a \times A \times B)$	$\rightarrow GEMM(a, A, B, 0, C)$
$(C := C + a \times A \times B)$	$\rightarrow GEMM(a, A, B, 1, C)$

Fig. 4. Rewrite system for optimizing to the GEMM function.

specification of instances of the Replacement pattern that follow a particular form. Consider the predicate P that verifies the semantics of a rewrite rule of the form $(x := f(x, y_1, y_2, ..., y_N), g(x, y_1, y_2, ..., y_N), P)$: P must ensure that x does not alias any y_i, requiring the author to repeat these semantic checks for every rule of this form. With the Assignment Replacement pattern, a pattern-based optimizer must either perform aliasing checks directly or transform the Assignment Replacement rule into an equivalent instance of the Replacement rule including the required checks.

What, however, happens if an actual expression does not quite match, even semantically, the left-hand side of an Assignment Replacement pattern?

4.2. The temporary removal adaptor

Consider an expression $z := a \times x + y$ that is similar to the semantic specification of AXPY, but is not semantically equivalent. In this case, two temporaries will be generated. It is possible, however, to remove one of these temporaries by executing $z := y$ followed by the procedure call $AXPY(a, x, z)$. Similarly, the expression $a \times x + y$ may be optimized into a call to AXPY depending on the nature of y. If y is a temporary value, overwriting it with another temporary value is reasonable assuming that y is not reused. In fact, the semantics of most programming languages does not support the direct reuse of temporaries, making this a reasonable assumption. An expression such as $a \times x + b \times y$ can therefore be optimized into $t := b \times y$ followed by a call to $AXPY(a, x, t)$. Generalizing the two examples, we introduce the Temporary Removal adaptor.

Definition. *Let (L, R, P) be an Assignment Replacement pattern where L is of the form $y := e$ for some variable y and expression e. We further assume $e[p] = y$ and $R[q] = y$ for some positions p and q. From this pattern the Temporary Removal adaptor produces the following new Replacement patterns:*

$$(z := e, (z := y,\ R[q \leftarrow z]), \quad P),$$
$$(e, \quad (\texttt{var}\ t = y,\ R[q \leftarrow t]), P).$$

where `var` $t = y$ denotes the declaration of a temporary variable t (local to the expression sequence) and its initialization to the value of y.

Applied to the just discussed AXPY Assignment Replacement, for example, the Temporary Removal adaptor generates the following two Replacement patterns:

$$(z := a \times x + y,\ \ (z := y,\ AXPY(a, x, z)), P),$$
$$(a \times x + y,\ \ (\texttt{var}\ \ t = y,\ AXPY(a, x, t)), P).$$

In the same way the Assignment Replacement used in the MTL library (see Fig. 3)

$$(C := A \times B + C, (A, B, C), P)$$

generates the two patterns

$$(D := A \times B + C,\ \ (D := C,\ (A, B, D)), P),$$
$$(A \times B + C,\ \ (\texttt{var}\ \ t = C,\ (A, B, t)), P),$$

and the GEMM Assignment Replacement (see Fig. 4)

$$(C := C + a \times A \times B, GEMM(a, A, B, 1, C), P)$$

the two patterns

$$(D := C + a \times A \times B, (D := C,\ GEMM(a, A, B, 1, D)), \quad P),$$
$$(C + a \times A \times B, \quad (\texttt{var}\ \ t = C,\ GEMM(a, A, B, 1, t)), \quad P).$$

5. Implementation

The implementation of an optimizer for the Replacement pattern and its subpatterns essentially requires the implementation of an expression rewrite system with rewrite rules supplied by the user. An immediate requirement of such a system is that the implementation of expression matching must be generic enough to support any form of expression, including user-defined operators (in the form of overloaded operators or function calls). Additionally, the user must be able to examine an expression to determine the semantics of the expression and its subexpressions to ensure correctness when applying a rewrite rule. Finally, the user must be able to construct new expressions to complete the rewriting step.

Simplicissimus consists of a conditional rewrite engine (implementing optimizations via the Replacement

pattern) and an extensible set of rewrite rules. The basic set of rewrite rules implements subpatterns of the Replacement pattern, including subpatterns based on algebraic structures (e.g.,

$$(x + 0, x, \text{right-identity}(+, 0)),$$

where right-identity$(\mathrm{Op}, \mathrm{Id})$ is a predicate defined to be true when Id is the right identity element for operator Op) and the Assignment Replacement and Temporary Removal subpatterns. Simplicissimus receives input from several sources: the basic set of rewrite rules (subpattern implementations), extension rewrite rules from some set of software libraries, library-defined mappings between Simplicissimus's internal representation and the library's C++ syntax, and, finally, the program source code. Simplicissimus transforms the program source code (available in the compiler's internal representation) into its own internal representation, applies rewrite rules that reduce the cost of an expression in a bottom-up fashion,[1] and transforms the result back into the compiler's internal representation.

The Simplicissimus optimizer can be viewed as a generalization of the simplifier component present in the vast majority of compilers. A compiler's simplifier performs simple rewrites based on known properties of built-in types, such as $x+0 \rightarrow x$ for integer values. Extending these simplifiers generally requires direct modification of the compiler source code, and is therefore not feasible for most users. Simplicissimus provides the rewrite capabilities of a traditional simplifier but is directly extensible via instances of the Replacement pattern.

We now discuss Simplicissimus's internal representation and implementation of new instances of the patterns presented here. Additional information regarding the expression rewrite process and the Simplicissimus architecture may be found in a separate paper [15].

5.1. Internal representation

Simplicissimus's internal representation consists entirely of C++ expression templates, a set of classes representing unary, binary, ternary, and other operations that are parameterized by the operators and operands, in a form similar to functional prefix form. Expression templates were discovered as an optimization technique for numerical computing [23] but have also been used for delayed evaluation and functional composition [6,10,11]. In Simplicissimus expression templates differ from most in that they have no runtime components: distinct variables and literal values are modeled as types, so that C++ expressions can be fully expressed as C++ types and manipulated at compile time.

Compile-time manipulations of expressions using expression templates have several advantages. They do not exist at run-time, so they incur no run-time overhead, and because they are constructed and manipulated entirely in C++ they are naturally compiler- and platform-independent, as discussed in Section 5.5. Most importantly, expression templates interact well with the complicated C++ type system and rely on *template metaprogramming* techniques [21] well-known in the C++ library design community. The drawback of expression templates is the inefficiency of the C++ template engine as an interpreter, although this problem is due not to any fundamental limitation of the C++ template engine but rather to design decisions made in current compilers.

The form of an expression template is similar to that of function prefix form. An expression $x + y * z$ can be expressed in prefix form as $(+\ x\ (*\ y\ z))$ and, similarly, as the expression template

$$\texttt{Expr} < \texttt{BinaryExpr} < \texttt{Add}, \texttt{X},$$

$$\texttt{Expr} < \texttt{BinaryExpr} < \texttt{Mul}, \texttt{Y}, \texttt{Z} >>>> .$$

Here we use the type names `Add` and `Mul` to represent addition and multiplication, respectively. Each operator or function will have a unique type (generally an empty class) that represents it in an expression template. Expressions are wrapped in class templates that contain the operator name and its operand(s), and are named based on the arity of the operation (`UnaryExpr`, `BinaryExpr`, etc.). The `Expr` class is a wrapper around each expression template that makes all expression templates easily distinguishable from other types.

The leaves of an expression tree – literal values and variables – are each expressed using unique types. The class template `Variable` is parameterized by the type of the variable (e.g., `int`) and by an integer identification number that is unique to that variable. In our example above, `X` may be `Expr<Variable<int,0>>` whereas `Y` could be `Expr<Variable<int,1>>`. Similarly, a class template `Literal` contains literal values, where a literal can be any C++ literal, but the notion has been extended slightly to include user-defined literals for abstract data types.

[1] Simplicissimus's default behavior is to compute a locally optimal solution for efficiency reasons, although the user may intervene with an alternative rewrite strategy that results in a globally optimal solution.

5.2. Matching expressions

Expression templates naturally lend themselves to pattern-matching via partial specialization. Partial specialization allows multiple definitions of class templates where each definition specifies the partial type structure of types it will be instantiated with. Expression templates use type structure to express expression evaluation, thus partial specialization can trivially be used to specify and match expressions. Figure 5 illustrates the *primary* template and one specialization of the class template `AXPYMatch`. The template can match any expression template via the primary template (the `valid` member will be `false`) but it can also match an expression $a*x+y$ where $+$ is represented by the type `VectorAdd` and $*$ is represented by the type `VectorScale`, in which case `valid` will be `true` to signify a match.

A library author defines the `AXPYMatch` class template (with specializations) based on the nomenclature of the library's domain. The `VectorAdd` and `VectorScale` types are names chosen by the library author to represent vector addition and vector scaling for that library, and are accompanied by transformation functions between the types and their corresponding C++ operations. The library-specific vocabulary is augmented by a base set of operations, such as assignment or comparison, that cross-cut library domains and are of general interest.

5.3. Semantic constraints

Semantic constraints determine whether or not a particular expression that syntactically matches the left-hand side of a rewrite rule will be semantically equivalent if the expression is rewritten. The check for semantic equivalence relies primarily on traits that describe the computational behavior of expressions, including which operands are modified, whether an operation has side effects beyond what is reflected in the operands and return value, and whether the operation is applicative (i.e., predictable given a set of operands and regardless of program state).

We will extend the expression matching class template `AXPYMatch` described in Section 5.2 to validate the semantic constraints of the AXPY subroutine in addition to matching the structure. This dual purpose is reasonable because semantic constraints are generally expressed as predicates based on the variables bound when matching the expression.

Figure 6 illustrates the validation of the semantic constraints on AXPY. Altogether three constraints on its parameters x and y, logically connected to the member `valid`, have to be met. For one, neither the evaluation of x nor the evaluation of y may have side effects, because the order of evaluation may change when rewriting an expression as a function call. The compile-time value of the member `has_side_effects` of any expression template is recursively determined using expression and user-defined operation traits. Additionally, x and y may not be the same variable. The `SameVariable` class template of Fig. 6 determines if the given expression templates are the same variable in the simplest case. A completely developed version of `SameVariable` is more extensive in that it takes into account user-defined operators that return references to one of their arguments, such as the C++ assignment operator.

5.4. Temporary removal adaptor

The optimizations described for temporary removal in Section 4.2 are implemented in Simplicissimus as a class template `InPlaceOperationSimp`. This class template is instantiated with a class template `T` that implements the functionality specific to an particular instance of the Temporary Removal adaptor. The functionality required of `T` is implemented by three members:

- `valid`: a boolean value that is true if and only if the syntactic and semantic constraints on the pattern are met;
- `result`: the type of the variable that is the target of the assignment in the underlying Assignment Replacement;
- `rewrite_with_target`: a class template that performs a rewrite of the given expression to the procedural form using the given target expression.

From the definition of `T`, `InPlaceOperationSimp` generates the two Temporary Removal rewrite rules defined in Section 4.2 along with the Assignment Replacement rewrite rule defined in Section 4.1. The `InPlaceOperationSimp` adaptor matches the syntactic forms $z := e$ and e, where z is a variable and e is an expression such that the `valid` member of `T<e>` is `true`, i.e., `InPlaceOperationSimp` delegates the pattern-instance–specific matching to the underlying domain-specific match template `T`. Once a match has been found, `InPlaceOperationSimp` will query `T<e>` to determine the subexpression

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

template<typename A, typename X, typename Y>
struct AXPYMatch< Expr< BinaryExpr<
                          VectorAdd,
                          Expr<BinaryExpr<VectorScale, A, X> >,
                          Y> > >
  { static const bool valid = true; };
```

Fig. 5. Using partial specialization to perform a syntactic match.

```
template<typename Expr1, typename Expr2> struct SameVariable
  { static const bool value = false; };

template<typename T, int ID>
struct SameVariable<Expr<Variable<T, ID> >,
                    Expr<Variable<T, ID> > >
  { static const bool value = true; };

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

template<typename A, typename X, typename Y>
struct AXPYMatch< Expr< BinaryExpr<
                          VectorAdd,
                          Expr<BinaryExpr<VectorScale, A, X> >,
                          Y> > >
{
  static const bool valid = !SameVariable<X, Y>::value
    && !X::has_side_effects && !Y::has_side_effects;
};
```

Fig. 6. Expressing the semantic requirements of the AXPY transformation using traits.

that would be overwritten by the rewritten operation, e.g., y in the expression $AXPY(a, x, y)$, allowing `InPlaceOperationSimp` to distinguish between the three rewrite rules mentioned without resorting to knowledge of the specific operations involved. When transformation is required, the subexpression is replaced with an appropriate expression via `rewrite_with_target`.

We complete the optimization of the AXPY function in Fig. 7 with our final implementation of the class template `AXPYMatch`. This class is to be directly used as the template parameter `T` of the `InPlaceOperationSimp` template to generate the rewrite rule class `AXPYSimp` that performs three temporary-removing optimizations within the Simplicissimus system: the AXPY Assignment Replacement along with the two optimizations generated by the `AXPYMatch` adaptor.

$(y := a \times x + y) \rightarrow AXPY(a, x, y)$,
$(z := a \times x + y) \rightarrow (z := y,\ AXPY(a, x, z))$,
$a \times x + y \rightarrow (\texttt{var}\ t = y,\ AXPY(a, x, t))$.

Partial specialization is again used to match AXPY's semantic constraints. The `valid` member is true whenever the expression is matched, and the target of the AXPY function is identified as y by the `target` member type. The actual rewriting into the more efficient form using AXPY is performed by the class template `rewrite_with_target`, which trivially builds an expression template using the ternary operation `AXPY`.

5.5. Integration in the GNU C++ compiler

Simplicissimus is a stand-alone optimizer written in the C++ template sublanguage, and is therefore natu-

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

template<typename A, typename X, typename Y>
struct AXPYMatch< Expr< BinaryExpr<
                          VectorAdd,
                          Expr<BinaryExpr<VectorScale, A, X> >,
                          Y> > >
{
  static const bool valid = !SameVariable<X, Y>::value
    && !X::has_side_effects && !Y::has_side_effects;
  typedef Y target;

  template<typename Z> struct rewrite_with_target
    { typedef Expr<TernaryExpr<AXPY, A, X, Z> > result; };
};
struct AXPYSimp : public InPlaceOperationSimp<AXPYMatch> {};
```

Fig. 7. Optimizations for the BLAS AXPY function based on the Temporary Removal adaptor.

rally compiler-neutral. Such a design allows optimizations based on Simplicissimus, such as the implementation of the Replacement pattern and its subpatterns, to be portable as well.

Integration of the Simplicissimus optimizer with a new compiler requires a transformation from the compiler's internal representation to Simplicissimus's expression templates for optimization, and then the reverse transformation to utilize the results of the optimization, both of which are defined in C++ by library authors. Within the GNU C++ compiler, approximately 2000 lines of C code were required to perform these transformations.

6. Related work

Design patterns [7] have been gaining wide acceptance as a tool for the construction and documentation of software systems, but their use does not generally extend beyond that of documentation or guidelines for programmers. The FRED [9] development environment, which extends this limited view of patterns to instead aid the programmer in the specialization of patterns for a particular purpose, thus shares our view that a design pattern is more than documentation or guideline. On the other hand, the goals are radically different from our own.

Tools for applying domain-specific transformations to optimize code, such as TAMPR [3] and Draco [13], enable authors of domain-specific languages to introduce optimizations based on the semantics of a particular domain. However, these general systems do not provide a conceptual framework for generating transformations that are common across multiple domains and multiple languages; that is, they do not take a pattern-based approach that describes optimizations as specializations of well-known, language- and domain-neutral optimization patterns. Constructing new, domain-specific languages that have similar optimization opportunities in other domains therefore results in a large amount of repetition.

Tools that allow library-specific optimizations within general purpose languages, such as the Broadway [8] open compilation system and the CodeBoost [2] source-to-source transformation system, enable users (library designers) to introduce additional semantic information and optimization opportunities for ordinary user code. Like domain-specific transformation, however, these systems give users little direction regarding optimizations that span multiple software libraries. Applying design patterns for optimization to any of these transformation systems would yield the same benefits as in our own Simplicissimus optimizer.

Work on the construction of *active libraries* [24], such as Blitz++ [22] and POOMA [14], has significantly narrowed the gap between library and compiler. Such libraries take an active role in the compilation process, tuning the generated code to specific tasks or specific architectures. Design patterns for optimization–or, specifically, implementations supporting them–can serve as a powerful tool for use by active libraries enabling optimizations that are impossible without such support. The Sophus C++ library [5] integrates with the aforementioned CodeBoost transformation system to apply domain-specific transformations to C++ code

that uses the Sophus library. The transformations there are similar to those of the Temporary Removal adaptor.

7. Conclusion

We have surveyed the design of several object-oriented numeric libraries with a strong focus on optimization techniques employed. From these designs we abstracted the common structure and semantics to form the Replacement pattern and two important subpatterns, the Assignment Replacement and the Temporary Removal adaptor. Additional patterns, such as the delayed element-wise transformation used by expression templates in libraries such as Blitz++ [22] and POOMA [14], are also known to exist but have not yet been studied.

Optimization patterns are not unique to the C++ language nor are they unique to the domain of numerical computing. For instance, transforming potentially expensive Java `String` concatenation expressions into equivalent operations on a temporary `StringBuffer` [1] is an instance of the Temporary Removal pattern. However, the usefulness of a particular design pattern for optimization depends greatly on the underlying domain.

Unlike many design patterns, the Replacement pattern and its subpatterns present optimization opportunities at a very high level of abstraction. Once instances of these patterns are identified, a compiler optimizer can attempt to generate better code based on strong, user-supplied assumptions on the semantic behavior of abstract data types.

We see these patterns as tools for advanced users and library authors to direct the optimization of high-level constructs that otherwise would be left unoptimized.

The Simplicissimus compiler optimizer implements the three patterns discussed in a compiler-independent manner. By using the strengths of the C++ language, Simplicissimus provides users with the ability to specify optimizations for abstract data types without requiring recompilation or additional extension of the compiler.

Acknowledgements

This work was supported in part by the National Science Foundation (NSF) NGS Grant 0131354.

References

[1] String concatenation and performance, http://developer.java.sun.com/developer/JDCTechTips/2002/tt0305.html, March 2002.

[2] O. Bagge, M. Haveraaen and E. Visser, CodeBoost: A framework for the transformation of C++ programs, Technical report, Universiteit Utrecht, The Netherlands, October 2000.

[3] J. Boyle, T. Harmer and V. Winter, The TAMPR program transformation system: Design and applications, in: *Modern Software Tools for Scientific Computing*, E. Arge, A. Bruaset and H. Langtangen, eds, Birkhauser, 1997.

[4] N. Dershowitz and J.-P. Jouannaud, Rewrite systems, in: *Handbook of Theoretical Computer Science*, J. van Leeuwen, ed., (Vol B), Formal Models and Semantics, Elsevier, Amsterdam, 1990, pp. 243–320.

[5] T. Dinesh, M. Haveraaen and J. Heering, *An algebraic programming style for numerical software and its optimisation*, Technical Report SEN-R9844, CWI, December, 1998.

[6] FACT! – *Multiparadigm programming with C++*, http://www. kfa-juelich.de/zam/FACT/start/index.html, 2001.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Massachusetts, 1994.

[8] S.Z. Guyer and C. Li, An annotation language for optimizing software libraries, in: *2nd Conference on Domain-Specific Languages*, T. Ball, ed, Usenix, 1999.

[9] M. Hakala, J. Hautamäaki, K. Koskimies, J. Paakki, A. Viljamaa and J. Vil-jamaa, Generating application development environments for Java frame- works, in: *Generative and Component-Based Software Engineering*, J. Bosch, ed., (Vol. 2186 of LNCS), Springer, September 2001, pp. 163–176.

[10] J. Jäarvi and G. Powell, *The Boost Lambda library*, http://www.boost.org/libs/lambda/doc/index.html, March 2002.

[11] J. Jäarvi, G. Powell and A. Lumsdaine, The Lambda Library: unnamed functions in C++, *Software–Practice and Experience* **33** (2003), 259–291.

[12] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Basic Linear Algebra Subprograms for Fortran usage, *ACM Transactions on Mathematical Software* **5**(3) (Sept. 1979), 308–323.

[13] J.M. Neighbors, The Draco approach to constructing software from reusable components, *IEEE Transactions on Software Engineering* **10**(5) (Sept. 1984), 564–574.

[14] POOMA. http://www.acl.lanl.gov/pooma/, 2001.

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

[16] V. Shoup, NTL: a library for doing number theory, 2001. http://www.shoup.net/ntl/.

[17] 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.

[18] J.G. Siek and A. Lumsdaine, The Matrix Template Library: Generic components for high-performance scientific computing, *Computing in Science & Engineering* **1**(6) (Nov.–Dec. 1999), 70–78.

[19] Simplicissimus, http://www.cs.rpi.edu/research/gpg/ Simplicissimus, 2001.

[20] The LiDIA Group, Lidia–a C++ library for computational number theory. http://www.informatik.tu-darmstadt.de/TI/LiDIA/.

[21] T. Veldhuizen, Using C++ template metaprograms, *C++ Report* **7**(4) (1995).
[22] T. Veldhuizen, Blitz++, http://www.oonumerics.org/blitz/, 2001.
[23] T.L. Veldhuizen, Expression templates, *C++ Report* **7**(5) (June, 1995), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.
[24] T.L. Veldhuizen and D. Gannon, *Active libraries: Rethinking the roles of compilers and libraries*, in Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98). SIAM Press, 1998.