Axioms as Generic Rewrite Rules
in C++ with Concepts

Xiaolong Tang and Jaakko Järvi

Texas A&M University
College Station, TX 77843, USA
x.long@infobeyonds.com, jarvi@cse.tamu.edu

Abstract

Compilers are typically hardwired to attempt many optimizations only on expressions that involve particular built-in types. Ideally, however, an optimizing compiler would recognize a rewrite opportunity for user-defined types as well, whenever the operands of an expression satisfy the algebraic properties that justify the rewrite. This paper applies the principles and techniques of generic programming and the “concepts” language feature, that was proposed to C++, to approximate this ideal. Concretely, a concept defines the signature and algebraic laws of a class of types. We attach rewrite rules to a concept, and by doing this make them applicable to the entire class of types that the concept defines. The annotation burden to a programmer is small—we take the existing declarations that a type models a particular concept as the annotation that enables generic rewrites. To apply generic rewrite rules, we instantiate them to type-specific rules. We use data-flow information from the compiler’s existing analyses to determine when these rules can be applied, and show how to interleave their application with function inlining to find more rewrite opportunities. Our prototype is implemented as an extension of the ConceptGCC compiler; our experiments show the approach is effective in eliminating abstraction penalties.

Keywords:
C++, concepts, rewriting, high-level optimizations

1. Introduction

Contemporary mainstream programming languages offer mechanisms for expressing high-level or domain-specific abstractions, and modern programs
take full advantage of them. Abstractions are, however, a barrier for an optimizing compiler. Rewrite rules for optimizations are typically hard-coded and apply only to “low-level” operations of primitive types. For example, one can expect a C++ compiler to substitute i for the expression i + 0 where the type of i is int. This transformation, \(i + 0 \rightarrow i\), is justified by the right identity law that holds for integers. This same identity law applies to the standard string type: assuming \(s\) is of type string, the transformation \(s + \text{string}(""") \rightarrow s\) preserves the meaning of the program. This is obvious to the programmer, but not to the compiler. If a compiler happens to rewrite \(s + \text{string}(""")\) into \(s\), it is not because of the right identity law. Instead, after inlining the constructor \(\text{string}(""")\) and the operator+() for strings, and the subsequent analysis of the expanded code may, after much work by the compiler, reveal the optimization opportunity.

Considering the economics of constructing industrial-strength compilers [1], it is not practical to extend compilers with rewrite rules for optimizing particular user-defined types. Type-specific optimizations thus tend to be limited to types in languages’ standard libraries, if any. For example, the valarray template in C++’s standard library [2] is specified to give guarantees of lack of aliasing. Few compilers take advantage of these guarantees.

This paper presents an approach where high-level rewrite rules are encoded, instead of for each type separately, as generic rules that apply to any type that satisfies a set of necessary requirements. For example, many standard simplification rules that compilers implement for primitive types are instances of more general algebraic rules. Our approach enables the specification of those generated rules and applies them equally to primitive types and to user-defined types. As an example, a single generic right-identity rule can justify the aforementioned transformations for int and string, as well as for an arbitrary number of other types.

Our approach is based on generic programming, a paradigm for developing reusable software libraries. A central element of generic programming is identifying oft-appearing algebraic classes in programs, and categorizing types according to these classes. Identifying such classes enables a concise definition of what is required of the parameters of a generic algorithm for the algorithm to work correctly and efficiently. In addition, types that belong to a certain algebraic class can also serve to establish the validity to carry out particular transformations for that type. We utilize the language mechanisms of generic programming to describe rewrite rules and justify their applicability. In particular, we utilize the “concepts” language feature [3] of
the C++ concepts proposal [4]\footnote{We use ConceptC++ to refer to C++ extended with concepts.} to specify rewrite rules for classes of types, and to establish which types belong to those classes. The result is an economical approach for specifying and enabling “high-level” optimizations for user-defined types, suitable for practical compilers—the annotation burden to the programmer is light and the implementation requires no major changes to the compiler architecture.

In simple rule-based rewriting, pattern matching is often confined to a single expression or statement; it fails to “see across a semicolon.” For example, consider the rewrite rule $f(x, g(y)) \rightarrow h(x, y)$. Its left hand side (LHS) matches an expression like $f(a, g(b))$, but a seemingly inconsequential transformation (adding a temporary variable) of the expression into \{t = g(b); f(a, t)\}, may hide the rewriting opportunity. Robison argues that optimizers should be robust for the above kinds of transformations [1].

Our strategy for robust high-level optimizations is to extend the high-level rewriting into the middle-end (ME) of the compiler, and reuse the ME’s existing analyses and transformations. We transform rewrite rules into conditional rewrite rules, so that we can exploit data flow analysis for more accurate matching. For instance, the above rewrite rule is transformed into the normal rewrite rule $f(x, t) \rightarrow h(x, y)$, but its application is guarded under the condition that $t$ is defined to be the result of $g(y)$. Such a rewrite rule with one or more augmented conditions is called a conditional rewrite rule, to be elaborated in 3.1.

Specifically, we make the following contributions.

- We demonstrate how generic programming supports the “lifting” of concrete type-specific optimizations to generic rewrite rules that can be instantiated to an open-ended set of user-defined types.

- We show how to utilize ConceptC++’s language features concepts, axioms, and concept maps for specifying generic rewrite rules and justifying their validity for desired user-defined types.

- We show how to utilize the compiler’s ME for high-level rewriting, including instantiating generic rewrite rules to produce type-specific rules to be applied by the ME, and controlling function inlining to avoid loss of rewriting opportunities.
• We provide an implementation of the approach as an extension to ConceptGCC [5]—Gregor’s extension of GCC to support concepts. We use the implementation to evaluate our approach, and show that our optimizer reduces the overhead of abstractions without a significant increase in compilation times.

2. Background: ConceptC++

This section briefly reviews the features of ConceptC++ relevant to this paper. The central language construct of ConceptC++ is concept. A concept defines requirements on a set of types. The requirements may be syntactic or semantic. There are three kinds of syntactic requirements: associated types, function signatures and nested requirements. An associated type is a type name that must be bound by each model of a concept. For example, the vertex and edge types might be associated types of a graph concept. A function signature expresses that a model must define an operation named and typed according to the signature. A nested requirement is a requirement on the associated types or parameters of a concept. Semantic requirements are expressed using the axiom [4] construct, as equivalences between expressions. The compiler is allowed to assume these equivalences hold for all models of a concept. We use the axiom feature for specifying generic rewrite rules.

A concept_map establishes that a type (or a tuple of types) models a concept, binding concrete types to all associated types, and providing implementations for the required functions. The concept system does not guarantee the satisfaction of the concept’s semantic requirements; assuring that a given model satisfies all axioms is the programmer’s responsibility.

Consider the example concept in Figure 1, corresponding to the algebraic structure monoid, simplified from a proposed taxonomy [6] of algebraic concepts for ConceptC++:

```cpp
concept Monoid<typename Op, typename T> : SemiGroup <Op, T> {
    T identity_element (Op, T); // identity element

    axiom Identity (Op op, T x) {
        op (x, identity_element (op, x)) == x; // right identity law
        op (identity_element (op, x), x) == x; // left identity law
    }
}
```

Figure 1: The Monoid concept
The two type parameters \( \text{Op} \) and \( T \) represent an operation and a set, respectively. The \texttt{identity\_element()} function corresponds to the identity element in a monoid, and the \texttt{axiom} specification defines monoid’s identity laws. By \textit{refining} the concept \texttt{SemiGroup}, the \texttt{Monoid} concept inherits all requirements defined in \texttt{SemiGroup}, notably that \texttt{Op} is an associative binary operation.

The integers \( \mathbb{Z} \) form a monoid with either the addition or the multiplication operator and a suitable identity element. The concept maps below establish these facts:

\[
\begin{align*}
\text{concept\_map} \ Monoid<&\text{plus}<\text{int}>, \text{int}> \\
\{ \text{int identity\_element (plus<int> op, int x) \{ return 0; } \}; \\
\text{concept\_map} \ Monoid<&\text{multiplies<int>}, \text{int}> \\
\{ \text{int identity\_element (multiplies<int> op, int x) \{ return 1; } \}; \\
\end{align*}
\]

Since C++ does not have first-class functions (operator symbols cannot be used as arguments to a concept), we use the established approach of \textit{function objects}, here \texttt{plus<int>} and \texttt{multiplies<int>}, to wrap operators into objects. The implementations of the \texttt{identity\_element} function define the desired identity elements for the two different models of \texttt{Monoid}.

As further examples, we could declare that a user-defined type \texttt{bigint} (for arbitrary precision arithmetic) and that the standard \texttt{string}, under appropriate operations, are also models of \texttt{Monoid}. We show the concept map for the latter:

\[
\begin{align*}
\text{concept\_map} \ Monoid<&\text{plus<string>}, \text{string}> \\
\{ \text{string identity\_element (plus<string> op, string x) \{ return std::string(""); } \}; \\
\end{align*}
\]

The above concept maps assert that the semantic requirements of \texttt{Monoid} hold for the established models. Thus, the identity laws hold, e.g., for the user-defined type \texttt{std::string}. We take advantage of such high-level properties of user-defined types to justify optimizations.

### 3. Generic and Flow-Sensitive Rewriting

The \texttt{axiom} feature of concepts offers a means to specify generic rewrite rules, applicable to all types that model a particular concept. The syntax of \texttt{axioms} can only express an equivalence between two expressions, not a rewrite rule. Therefore, additional conventions or annotations (e.g., with the help of C++11’s \texttt{attribute} mechanism) are necessary to specify a direction of applying a rewrite. In our prototype, we interpret each equation in an axiom
Table 1: Several models of Monoid. The triple of the values in the first three columns describes a particular monoid, and the fourth column shows the instance of the right identity rule in that monoid.

<table>
<thead>
<tr>
<th>Type</th>
<th>Op</th>
<th>Identity</th>
<th>Rewrite Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>+</td>
<td>0</td>
<td>i+0→i</td>
</tr>
<tr>
<td>int</td>
<td>*</td>
<td>1</td>
<td>i*1→i</td>
</tr>
<tr>
<td>set&lt;T&gt;</td>
<td>union</td>
<td>set&lt;T&gt;()</td>
<td>union(s, set&lt;T&gt;())→s</td>
</tr>
<tr>
<td>bool</td>
<td>&amp;&amp;</td>
<td>true</td>
<td>b&amp;&amp;true→b</td>
</tr>
<tr>
<td>string</td>
<td>+</td>
<td>string()</td>
<td>s+string()→s</td>
</tr>
<tr>
<td>bigint</td>
<td>+</td>
<td>bigint(0)</td>
<td>i+bigint(0)→i</td>
</tr>
<tr>
<td>matrix(m,n)</td>
<td>+</td>
<td>zero_matrix(m,n)</td>
<td>r+zero_matrix(m,n)→r</td>
</tr>
</tbody>
</table>

as a left-to-right rewrite rule. Consider again the Monoid concept in Figure 1. The two equations in the body of the Identity axiom result in the generic left identity rule \( R_1 \) and the generic right identity rule \( R_2 \).

\[
\begin{align*}
\text{op (identity_element (op, x), x)} & \rightarrow x & (R_1) \\
\text{op (x, identity_element (op, x))} & \rightarrow x & (R_2)
\end{align*}
\]

These generic rules express the common pattern of many type-specific rules. Table 1 lists several “non-generic” rewrite rules, all instances of the generic \( R_2 \) rule. In our approach, the instances are not hard-coded into the compiler, but instead generated from the generic rule of Monoid for all models of that concept. For example, the concept maps in Section 2 justify generating the left and right identity rules for \( \text{int} \) and \( \text{string} \).

3.1. Conditional Rewrite Rules

Directly mapping axioms to rewrite rules leads to rules with rigid patterns, unlikely to unveil many optimization opportunities. E.g., the rewrite rules in Table 1 directly match only to the abstract syntax tree (AST) of a single expression where the right-hand side (RHS) operand is the identity element; an operand equivalent to but not literally the same as the identity element would prevent the rule from being applied. Figure 2 illustrates the limitation using the rule \( x + \text{string}("") \rightarrow x \) as an example: the code fragment (a) can be transformed because one of its expressions contains an exact match against this rule’s LHS; the fragments (b) and (c) both have expressions that evaluate to a value equivalent to the rule’s LHS, but do not constitute a match.
string x ("text");
string y ("");
string z = x + y;

(a)

string x ("text");
string y ("");
string z = x + y;

(b)

string add (string a, string b)
{ return a + b; }

void main () {
    string y ("");
    string x ("text");
    add (x, y);
}

(c)

Figure 2: Code fragments that contain equivalent expressions to the LHS of the rewrite rule $x + \text{string}("\) \rightarrow x$.

To improve the robustness of the transformations, we generate rules with conditions. For example, the *conditional rule* for the right identity rule for string is defined as:

$$x + y \mid \{ \text{def}(y) = \text{string}("") \} \rightarrow x \quad (R3)$$

The transformation of a generic rewrite rule to a conditional rewrite rule is mechanical, and is explained in 3.2.1.

Applying the above rule requires the pattern matching to recognize the *operator*+() overload for string, followed by ensuring, based on data-flow information, that the definition for the operator’s second argument is equivalent with the result of the constructor call string(""). This strategy of rule application allows for rewriting the fragment (b) and (c) in Figure 2, as long as the compiler’s analyses are powerful enough to recognize the equivalences. In Section 3.2 we discuss how we utilize the compiler’s existing analyses and extend them towards this goal.

Rewrite rules with conditions are a widely applied technique.

3.2. Processing pipeline

Figure 3 illustrates the processing pipeline for effecting high-level optimizations. There are two new tasks for the compiler to perform: the generation of rewrite rules from axioms and the application of the generated rewrite rules to transform code. To accomplish the first task, the axioms in concepts are extracted during parsing and type-checking, then instantiated for specific types along template instantiation, and finally interpreted as rewrite rules to be stored into a “rule repository.” The rules in this repository are available
for the ME's rewrite engine. The second task is the responsibility of the
function abstraction analyzer, which governs function inlining and rule ap-
plication, as well as determines which rules to attempt in each function so
that excessive attempts of rule application is avoided.

3.2.1. Generating rewrite rules

We next illustrate the flow of generating conditional rewrite rules from
axioms in the pipeline of ConceptGCC.

To generate conditional rewrite rules from axioms, we exploit the process-
ing of concepts that is already part of ConceptGCC. Structurally, a concept
is very similar to a C++ class template. Indeed, ConceptGCC internally
represents concepts as class templates, and concept maps as specializations
(instances) of those class templates [5, 3]. Axioms in concepts are internally
represented with the AST nodes of a member function. Instantiating an ax-
iom for particular concrete types, as a result of processing a concept map
definition, is thus accomplished by way of normal template instantiation.
For example, the Identity axiom in Figure 1, when instantiated as part of a
concept map for Monoid<plus<string>, string>, yields this axiom instance:

```c
axiom Identity (plus<string> op, string x) {
    op (x, Monoid<plus<string>, string>::identity_element (op, x)) == x;
    op (Monoid<plus<string>, string>::identity_element (op, x), x) == x;
}
```

The qualifier Monoid<plus<string>, string>:: preceding the identity_element
function identifies the concept map for which the axiom is instantiated.

One rule in an axiom instance is a single expression. The pattern and the
condition(s) in a conditional rewrite rule are, however, composed of a group of
expressions that are chained together by data-flow information. For example,
the rule R3 contains two expressions, x + y and string(""), where y in the first
expression is defined by the second expression. In order to bridge the struc-
tural difference between a rule in an axiom instance and its corresponding
conditional rewrite rule, we recursively factor out the sub-expressions in the
rules of an axiom instance, substituting each sub-expression with a tempo-
rary variable. The result is a representation of the LHS and RHS expressions
of the rules in a three address form. In GCC this form is called GIMPLE [7].
Then we conduct flow analysis and necessary transformations, described be-
low, to obtain the end result, the conditional rewrite rules corresponding to
the axiom instances. In detail, we apply four steps of transformations to an
axiom instance.

8
Figure 3: The processing pipeline for effecting high-level optimizations. The boxes with single lines represent functional units related to processing of high-level rewrite rules. The boxes with double-lines represent functions that are part of a typical compiler—we only show the functions relevant to our framework. The arrows indicate data dependencies (flow of data) between functional units.
**Step 1** extracts rewrite rules from an axiom instance. As a function is the basic processing unit in a compiler from parsing to code generation, we represent each rule in the axiom instance as a pair of functions, extracted and derived from each side of the rule by the rule extractor unit. In this discussion, we prefix the names of such rule functions with “rule”. As an example, the first rule in the above axiom instance of Identity gives rise to this pair of rule functions:

```c
string rule_string_identity_lhs (plus<string> op, string x) {
    return op (x, Monoid<plus<string>, string>::identity_element(op, x));
}

string rule_string_identity_rhs (plus<string> op, string x) { return x; }
```

**Step 2** translates a rule function into its intermediate representation and performs the subsequent control and data flow analyses for it. This step is part of the normal processing for a function in the compiler. E.g., subject to GCC’s standard translation into the GIMPLE form [7], the above rule functions turn into:

```c
string rule_string_identity_lhs (plus<string> op, string x) {
    t1 = Monoid<plus<string>, string>::identity_element(op, x);
    t2 = op(x, t1);
    return t2;
}

string rule_string_identity_rhs (plus<string> op, string x) { return x; }
```

Using this form, the compiler produces the control and data flow information for the just generated rule functions.

**Step 3** eliminates the extra function abstractions that are present in an axiom instance but should not be used in rewrite rules generated from the axiom instance. For example, the Identity axiom in the Monoid concept is written in terms of the identity_element function. Each Monoid concept map specializes this function to some expression that will construct an identity element for a particular model of a Monoid; it is this specialized expression that is used in non-template user code, not the identity_element function. In the case of Monoid<plus<string>, string>, the identity_element function is specialized to the expression string(""").

Another source of extra abstractions are function objects that wrap function symbols, such as plus<string>() in the model Monoid<plus<string>, string> to wrap the call to the operator+ function overloaded for string.
The rule de-abstractor unit identifies local functions in concept maps and/or the function calls resulting from the use of function objects, and applies function inlining to eliminate these abstractions. In our running example, the rule functions shown in Step 2 become:

```c++
string rule_string_identity_lhs (plus<string> op, string x) {
    t1 = string("");
    t2 = operator+(x, t1);
    return t2;
}

string rule_string_identity_rhs (plus<string> op, string x) { return x; }
```

**Step 4** constructs the rule’s LHS and RHS patterns from their corresponding rule functions, and puts them into the rule repository. The pattern of each side of a rule is an AST-like tree, where the intermediate nodes represent function calls, or unary or binary operations, and the leaf nodes constants or free variables, i.e., function parameters. To construct such a pattern, we examine the control flow graph (CFG) of a rule function. We assume that data-flow analysis has computed the use-definition information for the CFG. Starting from the last expression in the CFG (i.e., the argument of the return statement in the rule function), we recursively process each expression as follows. If an expression represents a function call (or an operator invocation), we create a pattern node for this call and link the arguments’ patterns (constructed recursively) to the node; if an expression is a temporary (all temporaries are generated by the compiler), we replace this temporary with the pattern constructed by processing its definition in the CFG; if an expression is a constant, the constant is the pattern. As an example, consider the rule function `rule_string_identity_lhs`. Figure 4(a) shows its CFG. Applying the pattern construction strategy on its CFG produces the tree in Figure 4(b), where x is a free variable and "" a constant. This tree corresponds to the LHS of the rule \( \mathcal{R}3 \).

### 3.2.2. Applying Rewrite Rules

To apply a rule to a function, we downward traverse the function’s CFG, and employ the following strategy to each statement of the function. Given a statement, we match the rule’s pattern against the statement’s AST. In this matching process, the pattern is represented concretely as a tree data structure, whereas the CFG implicitly defines a tree of an expression or a statement through the use-definition relation. When we try to match a
Figure 4: The CFG for the rule `string_identity_lhs` rule function (a), and the pattern derived from it (b).

subexpression of the pattern to a variable in the CFG, we determine whether the variable is defined in the CFG. If it is not, the match fails. If it is, the variable is replaced by the expression defining it, and we try to match the subexpression to this expression recursively. A successful match is followed by replacing the redex denoted at the returned match position in the statement’s AST with the appropriately substituted RHS of the rule.

The above strategy is sufficient, e.g., to uncover the rewriting opportunities in the code in Figure 2(a) and 2(b). To reveal the rewriting opportunity in the code in Figure 2(c), we integrate our rewriting approach with function inlining.

3.2.3. Transformations and Inlining

Inlining plays two contradictory roles in application of rewrite rules. On the one hand, inlining can expose new facts justifying rewriting. E.g., inlining `add()` in Figure 2(c) reveals the fact that `def(b) = string("")`, which justifies the application of the rule $R_3$. On the other hand, too early inlining may lead to the loss of rewriting opportunities. For example, in Figure 2(c), if the constructor `string()` or the function `operator+()` is first inlined, the rule $R_3$ no longer matches. Thus, a strategy for interleaving inlining and rule application is necessary.

Compilers tend to carry out inlining in one pass, choosing candidates (functions to be inlined) and inlining them all at once. It would be prohibitively expensive to try a large number of different inlining orders, and with each inlining step, attempt to apply rewrite rules. To help reduce the search space of potentially useful inlining orders, we apply function abstrac-
tion analysis to obtain a measure, the abstraction index, of how “abstract” each function is in relation to other functions.

The abstraction index of a function is obtained from the program’s call graph. Built-in functions and operations are at the lowest abstraction level, and a (non-recursive) function is always on a higher abstraction level than any of its callees. Concretely, the abstraction index $\phi$ of a built-in function is 0. For a non-recursive, user-defined function $f$, it is defined as:

$$\phi(f) = \max(\phi(g_1), \ldots, \phi(g_n)) + 1,$$

where $g_i$ are the callees of $f$ (and no $g_i$ is $f$).

We compute the value of $\phi$ for a given function in the depth-first traversal order of the program’s call graph. To deal with recursive functions, we keep track of what functions have been visited and ignore recursive calls. More precisely, if a call path leads to a cycle from a caller to itself, then none of the nodes in that call path contribute to the computation of the abstraction index. The net effect is that recursive calls to a function have no effect on computing the function’s abstraction index. For example, in the code in Figure 2(c), assuming the abstraction indices of the constructor `string()` and function `add()` are, respectively, 1 and 2, then $\phi(\text{main}) = \max(\phi(\text{string()}), \phi(\text{add})) + 1 = 3$.

The functions in a program, in particular those identified as candidates for inlining, can be partitioned based on their abstraction indices. Starting from functions in the partition with the highest abstraction index, we inline each candidate, and attempt to match the rewrite rules in the body of the just inlined function. The process is then repeated recursively for the functions in the partition with the next lower abstraction index. With this strategy, the rewriting effort becomes proportional to the number of partitions.

To further improve the efficiency of applying rewrites, we use the abstraction indices of functions to guard rewriting. Naively, each rewrite rule could be attempted to each function. In practice this is not necessary. We can rule out many rewrite rules based on their rule functions’ abstraction indices.

Given a function $f$ and a rule function $r$, if $\phi(f) < \phi(r)$, the rule corresponding to the rule function $r$ cannot match an expression in $f$. From this property, we derive a practical approach to combine inlining and rewriting.

Our approach consists of one preprocessing and two rewriting phases. In the preprocessing phase, we partition all functions that are candidates for inlining according to their abstraction index, and do the same to all rewrite
rules. We order the partitions of the function candidates and the rewrite rules, respectively, in the descending order of their abstraction indices. In the first rewriting phase, then, we attempt to apply each rewrite rule to each function whose abstraction index is at least that of the rule function. In the second rewriting phase, we iterate over the ordered partition of the rewrite rules, interleaving inlining and rewriting operations. Specifically, in each iteration, we use the rewrite rules from the partition as potential rewrite rules, inline those function candidates whose abstraction indices are greater than or equal to the abstraction index of the potential rewrite rules, and attempt to match the potential rewrite rules to the bodies of the just inlined functions.

The above approach is practical, as demonstrated by our experiments described in Section 4. The number of iterations in the second phase is limited by the highest abstraction index of any of the rule functions, and each iteration has a set of rules to apply that is disjoint from the sets of other iterations.

3.3. Discussion

The way we generate rewrite rules from axioms applies in principle to any axiom, but of course not all axioms define worthwhile transformations. Therefore, presently we only process a select set of concepts and axioms, specifically the algebraic concepts. There is no technical difficulty to offer a general interface for configuring the concepts and axioms to be used as sources of rewrite rules.

Our approach does not provide any checks of confluence, correctness, or termination of the rewrite system generated. Users are assumed the responsibility to define semantically correct rewrite rules, and a maximum number of rewriting attempts rules out non-termination.

One strength of our approach is the incorporation of data-flow information in the rewrite system. Data-flow information provides the links between expressions that enable pattern matching across statement boundaries. Section 3.2.1 describes how to exploit use-definition information for pattern construction and pattern matching. In object-oriented programs, however, data-flow analysis tends to provide more general information than use-definition knowledge, i.e., what is known as mod/ref information [8]. Mod/ref information describes what objects are modified or read at each program point. Mod/ref information extends the scope of our approach to a larger set of rewrites. Our previous examples use a single expression to specify the rewrite
pattern (i.e., the LHS of an equation in an axiom). More general rewriting, however, requires the ability to express a pattern of which only part is rewritten. For this purpose, we use the C++ comma operator. Of a sequence of expressions grouped by a comma operator, we view the rightmost expression as the rewrite target, and the other expressions as the context where the rightmost expression is to be rewritten. As an example of this kind of an axiom, consider the following:

```cpp
class ListContainer<
  typename C>
{
  // ...
  axiom Idempotence(C c)
  { (c.sort(), c.sort()) == void(); }
}
```

We assume the `ListContainer` concept represents a concept of types each of which defines a `sort()` member function. E.g., `std::list` would be a model of this concept. From the above axiom, we derive the rewrite rule “for an instance of a type modeling the `ListContainer` concept, if, in a basic block of a CFG, a `sort` function call acting on the instance is preceded by a second `sort` call on the same instance, and there are no modifications to the instance between the two `sort` calls, it is safe to eliminate the second call.” It is through mod/ref analysis that this kind of information is obtained.

In order to exploit mod/ref information, we generalize our strategy for pattern construction—function parameters no longer need to be leaf nodes in a pattern tree. Concretely, we add a new subcase to the cases for processing an expression (in Step 4 in Section 3.2.1): when an expression is a non-temporary variable, i.e., a function parameter, processing such an expression falls into two sub-cases; if a statement dominates the expression and either defines, modifies, or accesses this variable, we create a pattern node for the variable and link it to the pattern created from the statement; otherwise, as before, the pattern node created for the variable is a leaf node.

The above `Idempotence` axiom has, of course, little practical use. The examples below demonstrate more useful common transformations that the above adjustment enables.

*Copy propagation.* This optimization replaces the occurrences of the target of an assignment with its source. It is traditionally only applied to built-in types, but many commonly used user-defined types have semantics that make copy propagation a safe optimization. Such types are often said to conform to *value semantics*. Dehnert and Stepanov investigate the axiomatizations that guarantee value semantics of user-defined types; these axiomatizations
generalize the semantics possessed by built-in types like `int`, `float`, etc., in terms of the fundamental operators of constructing, copying, assignment, and comparing for equality [9]. A type that supports the fundamental operators with these axiomatizations is a regular type. In ConceptC++, the Regular concept describes the requirements for a regular type [10, §20.1.7]. The rewrite rule of copy propagation is captured with the following axiom:

```cpp
concept Regular< typename T > : ... { 
...
  axiom CopyPropagation(T& x, T& y, T& z) 
  { (y = x, z = y) == (z = x); } 
}
```

With this axiom interpreted as a rewrite rule, copy propagation optimization applies to all regular types. For example, in the code below, assuming that the type `T` is a model of the Regular concept, the `c = b` expression is rewritten into `c = a`.

```cpp
T a, b, c;
...
b = a;
... // No modification to a and b
c = b;
```

To see how copy propagation works, consider the pattern tree resulting from the LHS of the equation in the CopyPropagation axiom as shown in Figure 5; `x`, `y`, and `z` are free variables. Pattern matching occurs in the basic block of a program’s CFG. A potential match identifies two expressions in a basic block, and a range of expressions in between. If data flow analysis can show that the variables in those two expressions are not modified in the range of expressions in between, the match is successful.

4. Evaluation

We evaluated both the optimizing effectiveness of our approach and the impact it has on the compiling effort. Our prototype, implemented as an extension of the ConceptGCC compiler, can be obtained from our project home pages [11]. The prototype adds a command line option “-fconcept-simplify” for users to switch on the concept-based optimizations. In our test runs, we disabled the exception mechanism with “-fno-exceptions” and enabled optimizations by “-O2”.

16
The former is to reduce the implementation effort of our prototype; we do not foresee any fundamental difficulties in adding support for exceptions. The evaluation platform was an iMac 2GHz Intel Core Duo, running Mac OS X 10.5.3.

To measure the effectiveness of our approach, we selected programs that contain algebraic expressions that an optimizing compiler routinely simplifies if the arguments of those expressions are of built-in types. We replaced the built-in types with user-defined types whose operations obey the same algebraic laws that justify the simplifications on built-in types, and measured the abstraction penalties of the codes. Abstraction penalty is defined as the ratio of the execution time of an abstracted implementation over a direct implementation [12, §D.3]. The test programs are from Adobe’s C++ performance benchmark suite\textsuperscript{2} [13], designed to measure, among other traits, abstraction penalties of C++ compilers.

The benchmark wraps a varying number of double values into user-defined classes that support arithmetic operations (and thus follow the same algebraic rules as double) and executes code that repeatedly evaluates arithmetic expressions on objects of those classes. Figure 6 summarizes the results. The user-defined classes, DoubleClass, Double2Class, and Double4Class wrap one, two, and four doubles, respectively. Each test was repeated for each of these classes. The names of the tests indicate the algebraic operations being tested.

\textsuperscript{2}The benchmark suite’s current public release does not yet include the tests we used; we make the tests available on our project home pages [11].
Figure 6: The benchmark results for algebraic simplifications for user-defined types. D1, D2, and D4 denote DoubleClass, Double2Class, and Double4Class, respectively. The columns denoted with (A) show the abstraction penalties measured with high-level optimizations on, the columns (B) show the same measured with those optimizations off.

As an example, the “mixed algebra” tests measure the optimizer’s efficiency for compound expressions that include more than one kind of algebraic operations. The code for these tests, where $T$ is a placeholder for one of the above three user-defined classes, is as follows:

```cpp
T test (T input) {
    return (- (T(0) - (((input + T(0)) - T(0)) / T(1))))) * T(1);
}
```

The baseline for the test of abstraction penalty measurement in this case is the function $T$ base (T input) {

```
return input;
```
}

The abstraction penalty is consistently one with our approach. With our optimizations inactive, on the other hand, the compiler only got rid of the abstraction overhead in three of the twelve cases. We also measured the impact of our approach on compilation time: for this benchmark, the compilation time increased by a factor of 1.0035.

What is the burden for the programmer to enable the identity rules for the user-defined types used in the benchmark? The identity rules are defined in the concept Monoid, which is predefined in a header file. The necessary concept map for the additive monoid for DoubleClass below is an example of one of the nine concept maps (an additive, subtractive, and multiplicative monoid for each of DoubleClass, Double2Class, and Double4Class) that the programmer would write to enable the identity laws:

```cpp
concept_map Monoid<plus<DoubleClass>, DoubleClass> {
    DoubleClass identity_element (plus<DoubleClass> op, DoubleClass x)
}
```

---

3Note that strictly speaking IEEE doubles do not form a monoid because of the special NaN and Inf values.
To measure the impact of high-level simplifications to later analysis and optimization passes of the compiler, we estimated the size of the compiler’s intermediate data at various stages of the compilation by measuring the size of the output GCC generated for each compilation stage when invoked with the option `-fdump-tree-all`. The program we compiled was the above benchmark. Results are shown in Figure 7. In the beginning, annotations for high-level optimizations, concepts and concept maps, account for a larger size of the representation, but during further stages, the size decreases: high-level rewrites, applied early, reduce the workload of the later phases of the compilation.

![Figure 7: The impact of early high-level transformations on the size of intermediate data throughout compiling our benchmark. The horizontal axis enumerates the compilation passes in chronological order, the vertical axis denotes intermediate data size in kilobytes. The dashed line was obtained with `-fconcept-simplify`, solid line without it.](image)

We also measured the performance of the example code in Figure 2(c), where appropriate inlining strategy is necessary to uncover the optimization opportunity. We measured the execution times of a loop invoking `add(x, y)` repeatedly (100,000 times). The running time with the high-level optimizations turned on was 0.013 seconds, compared to the 0.015 seconds when they were turned off. To exercise the left identity rule, we repeated the experiment for the call `add(y, x)`. Now the running times were 0.013 with high-level optimizations, and 0.033 without.
5. Related Work

A compiler supporting user-extensible high-level optimizations has been approached from several directions. *Active libraries* are a lightweight “non-invasive” approach, where optimizations are defined in a library. For example, the C++ template mechanism provides rudimentary support for active libraries and user-defined optimization rules [14].

A general rule-based transformation system lets users compose rewrite rules into transformations and thus design domain-specific language transformation tools. One such general transformation system is Stratego [15]. Stratego underlies the implementation of the CodeBoost transformation system for C++ by Bagge et al. [16]. CodeBoost supports annotating rewrite rules with conditions, so its rewriting ability is more powerful than just pattern matching on ASTs. Compared with CodeBoost, we do not provide a means to specify arbitrary conditions for rewrite rules; instead, rewrite rules are generated from axioms, and their conditions are data-flow facts. CodeBoost transformations are defined per class, whereas in our approach rules are applicable generically to all types that model specific concepts.

Olmos et al. introduce “dynamic rewrite rules” to model run-time data-flow facts, which allow for context-sensitive transformations and the combination of transformations and analyses [17]. Another rule-based system is TAMPR [18], which has been used to define high-level transformation rules for calls to the routines of the LINPACK library.

Several approaches support extending a compiler with new rewrite rules. Simplicissimus by Schupp et al. [19] shares the idea of generic rules with our approach. Their work preceded the introduction of the built-in “concepts” language constructs of ConceptC++, and hence they used other means to describe concepts and to organize rewrite rules. Simplicissimus represents rewrite rules internally using expression templates, and exploits C++ template processing to effect those rules. This is a slow mechanism, and limited to the front-end of the compiler: rewrite opportunities discovered as a result of flow analyses cannot be applied. In our earlier work [20], we developed an approach motivated by Simplicissimus, but realized in ConceptC++. Similar to Simplicissimus, this work is limited to flow-insensitive rewriting: rewrite rules match only in generic functions where the “concept constraints” to justify rewrites are immediately visible.

Peyton-Jones et al. extended a Haskell compiler with the ability to apply user-defined rewrite rules [21]. Like us, they also faced the problem
of integrating function inlining and rule application; they addressed it by giving users the ability to control in which compilation phases a particular rule should be applied. Recently, Willcock et al. proposed concept-enabled generic program analyses and transformations [22]. Our work can be viewed as a practical realization of the general approach they advocate.

The Broadway compiler [23] can support sophisticated domain-specific transformations, targeted to the domain of numerical computations. An annotation file can describe detailed characteristics of a library routine, its effect on the data it manipulates, and transformations or actions to take when the data satisfies stated conditions. Cobalt by Lerner et al. explores temporal logic for specifying analyses and transformations [24], whereas Deepweaver by Falconer et al. relies on a Prolog-based query language for this purpose [25].

In the context of C++, ROSE allows for restructuring the ASTs of a program, e.g., to parallelize the operations on arrays [26]. Transformations in ROSE are described in terms of ASTs and require quite some expertise to define. OpenC++ extends C++ with a mechanism to define new syntax, new annotations and new object behavior [27], and can thus also be used for implementing user-defined optimizations. Edelsohn puts forward a proposal [28] to extend C++ with hooks in classes that would enable the expression of user-defined optimization rules. The goal was to allow, for example, generating efficient code (avoiding the introduction of temporary variables) for statements like

\[ A = B \times C + \text{++}D/E \]

where all variables are of a particular array type.

6. Conclusion and Future Work

We engage generic programming and the “concepts” language feature of C++ for realizing generic user-extensible simplifications. The burden to use these optimizations is small. A programmer specifies generic rewrite rules with axioms in concepts, and the rules are put to use for a particular type by simply stating that the type models a particular concept.

We show that the ability to perform high-level user-defined optimizations can be built into an industrial strength compiler without distracting the compiler architecture in major ways, and that the increase in the compiling resources to support these optimizations stays small. Further, generic rewrite rules that apply to a large class of user-defined types effectively eliminate abstraction penalties where standard low-level optimization techniques fail to do the same.
In the future, we plan to explore a larger class of optimizing transformations, including domain-specific optimization rules, that could be encoded using ConceptC++’s axioms, and thus put to work using our framework. In particular, we plan to investigate the ways of leveraging the associativity, transitivity, and commutivity laws. Further, our reliance on the compiler’s existing data-flow analyses is a compromise. On one hand it is economical, as we avoid extending the compiler with new analyses, but on the other hand we necessarily lose precision as the data-flow analyses that operate on the compiler’s intermediate representation cannot deliver fully accurate results—some relevant abstractions have already been “compiled away” at earlier stages of compilation. We are currently developing a data-flow analysis that operates directly on objects and operations of user-defined types. By utilizing the high-level properties, in particular regularity, we obtain increased precision.

Our practical approach to optimization is positioned between the generally applicable but almost exhausted low-level optimization opportunities, and specialized but costly domain- and library-specific compilers and optimizers.

References


URL http://www.open-std.org/jtc1/sc22/wg21


URL http://www.open-std.org/jtc1/sc22/wg21


