Axiom-Based Transformations: Optimisation and Testing

Anya Helene Bagge [1] Magne Haveraaen [2]

Department of Informatics
University of Bergen,
Norway

Abstract

Programmers typically have knowledge about properties of their programs that aren't explicitly expressed in the code – properties that may be very useful for, e.g., compiler optimisation and automated testing. Although such information is sometimes written down in a formal or informal specification, it is generally not accessible to compilers and other tools. However, using the idea of *concepts* and *axioms* in the upcoming C++ standard, we may embed axioms with program code. In this paper, we sketch how such axioms can be interpreted as rewrite rules and test oracles. Rewrite rules together with user-defined transformation strategies allow us to implement program or library-specific optimisations.

Key words: rewrite rules, axioms, optimisation, testing, C++, concepts

1 Introduction

In the coming C++0x standard [3] [10], it is proposed that axioms be part of the new *concept* construct. The idea of concepts is to let programmers place restrictions on template parameters. For instance, a generic sorting function may specify that its argument should be an *Indexable* object with *LessThanComparable* elements. Without concepts, one would have to just go ahead and use the indexing and less-than operators, and then the compiler would give an incomprehensible error message if someone tried to use the sorting function with an unsuitable data structure.

Concepts can also be organised in a hierarchy – allowing well known algebraic concepts to be mapped to C++ concepts [8] in a structured way.

[1] `http://www.ii.uib.no/~anya/`
[2] `http://www.ii.uib.no/~magne/`
[3] When we refer to 'C++' in this paper, we refer to the concept-enabled proposed standard.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

Using a *concept map*, one can specify that a class or group of classes satisfies a given concept, and possibly also map between the names and signatures of the concept and those of the class. For example, "my class is *LessThanComparable*, with '$!(x >= y)$' as the less than operator". The idea of axioms in the proposed standard is in the early stages. There is a defined syntax for it, and a few examples, but the specification of behaviour is fairly limited.

The compiler is allowed, but not required, to replace expressions with equivalent expressions according to axioms – for optimisation purposes, for example. Earlier, the compiler had little opportunity to make assumptions about user code, for instance, it could not apply the many simplifications available for built-in operators to expressions with user-defined operators. Such rules may now be stated as concepts, but there is still no way to give hints to the compiler as to which axioms may be useful, which side of the equality is preferable, how rewrite rules should be applied, etc.

Axioms may serve many purposes in a software system:

(i) Recording knowledge about template arguments for generic classes and methods.

(ii) Proving that one set of axioms implies another set of axioms – for example, that a class belonging to a concept *C1* may be used where a concept *C2* (with the same or a smaller signature) is required – and also program verification – proving that a class satisfies its stated properties.

(iii) Semantics-preserving rewrites of the code, e.g., for optimisation purposes.

(iv) Testing that a class satisfies its stated properties.

Item *(i)* is more or less where the proposed standard stands today; with axioms as a form of structured documentation of the requirements on concepts. Item *(ii)* may not be applicable to C++, partly because C++ syntax and semantics is very difficult to analyse, partly because many requirements cannot be expressed as axioms. We will focus on items *(iii)* and *(iv)* in this paper, based on our previous experience with user-defined rewrite rules for C++ [1] and axiom-based testing [12].

The rest of this paper is organised as follows. We start by introducing axioms, then discuss the use of axioms for rewriting (Sect. 3) and testing (Sect. 4). We will then sketch some implementation issues (Sect. 5), and finish with a discussion and conclusion (Sect. 6).

2 Concepts and Axioms

The proposed C++ standard syntax for axioms is shown in Fig. 1. Each axiom definition can contain multiple axioms, and the axioms themselves are expression statements with the '`==`' operator. For example, here's a concept *Monoid* with an *Identity* axiom (taken from [10]):

```
concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
```

```
RequiresClause? "axiom" Identifier "(" ParamDecls ")" AxiomBody
                                        -> AxiomDef
"{" Axiom* "}"                          -> AxiomBody
ExpressionStmt                          -> Axiom
"if" "(" Condition ")" ExpressionStmt -> Axiom
```

Fig. 1. Proposed C++0x Standard Syntax for Axioms (in SDF2 notation). The *RequiresClause* allows for additional concept constraints on axiom parameters – we will ignore it in this paper.

```
RequiresClause? "axiom" Identifier "(" ParamDecls ")"
  GroupClause? AxiomBody                -> Declaration
":" {GroupName ","}+                    -> GroupClause
"{" (Axiom|Statement)* "}"              -> AxiomBody
"assert" "(" Expression ")" ";"         -> Axiom
```

Fig. 2. Our syntax for Axioms (in SDF2 notation).

```
  T identity_element(Op);
  axiom Identity(Op op, T x) {
    op(x, identity_element(op)) == x;
    op(identity_element(op), x) == x;
} }
```

We may then specify that a class *Vector* satisies the *Monoid* concept, with *Vector::plus* [4] as the operation and *Vector::zero* as the identity element:

```
concept_map Monoid<Vector::plus, Vector>{
  Vector identity_element(Vector::plus) {
    return Vector::zero;
} }
```

For our work, we have chosen a slightly different syntax (see Fig. 2). Our syntax extends the original syntax by allowing *axiom groups* (see Sect. 2.1), by allowing any statement to be used within the axiom body (used in testing, see Sect. 4), and by marking the actual axiom with the keyword *assert*. Also, we allow axioms to be declared outside of concepts, so that one may attach simple axioms directly to a class, instead of having to declare concepts and concept maps for it.

A standard C++ concept map is only 'active' when the mapped class is used as a template argument that has been constrained to a concept. E.g., *Vector* will only be considered a *Monoid* when it is used as a *Monoid* in generic code. This restricts the use of axioms, and we will instead assume that the

[4] *Vector::plus* is a class wrapper around the plus operation, necessary to use it conveniently as a template argument. *Vector::plus()* is an object of this class, usable as a function as it has an overloaded `()`-operator. This is a usual way of doing things in C++, and we shall refrain from commenting on the intuitiveness of it...

```
GroupDef                                          -> Declaration
"axiom_group" GroupName (GroupBody | ";")         -> GroupDef
"{" GroupDecl* "}"                                -> GroupBody
"using" Name ";"                                  -> GroupDecl
AxiomDef                                          -> GroupDecl
TemplateSpec GroupDecl                            -> GroupDecl
```

Fig. 3. Syntax for Axiom Groups. A *GroupDef* defines or extends a named axiom group. The *GroupName* is an identifier or a qualified (nested) name. The *GroupBody* lists the axioms or axiom groups that form the group. *Name* should be the name of an axiom, group or concept.

existence of a concept map means we can use concept axioms for the classes and operations in the map.

Using the *assert* keyword makes it easier to allow a wider variety of statements in the axiom body, and to allow for other kinds of axioms than just equality. It also simplifies making the axioms executable as test code (Sect. 4).

In our syntax, the above monoid concept becomes:

```
concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
  T identity_element(Op);
  axiom Identity(Op op, T x) : simplify {
    assert(op(x, identity_element(op)) == x);
    assert(op(identity_element(op), x) == x);
} }
```

The *axiom group* '*simplify*' identifies the *Identity* axiom as usable for a simplification rule, where the right-hand side is assumed to be simpler (less resource-intensive) than the left-hand side.

2.1 Axiom Groups

Axiom groups are used to identify which axioms are useful as rewrite rules, and to distinguish between different types of rules – e.g., simplification rules, reordering rules like associativity/commutativity, or rules that should be applied early or late in the transformation process [5]

Axiom groups are defined using the *axiom_group* construct (Fig. 3), or by listing the group name in the *group clause* of an axiom definition. The *axiom_group* definition is open-ended and can be extended later on. Here's a sample axiom group *simplify*, containing the *Identity* axiom from the *Monoid* concept (having the same effect as in the example above):

```
axiom_group simplify {
  template<typename Op, typename T>
    using Monoid<Op,T>::Identity;
```

[5] Such tagging is also important in specification and proof systems like CASL [16].

```
}
```

The template declaration has the effect of universally quantifying the operator and type for the monoid, adding the identity axioms from all monoids. [6]

The *using* directive allows us to add a single axiom, all axioms from a given concept (e.g., *using Monoid*), or all axioms from another axiom group (*using my_simplify;*). Axioms may also be defined directly in the axiom group.

Listing an axiom in a group definition is equivalent to listing the group name in the axiom definition. We allow both possibilities to cut down on the amount of code that needs to be written for a simple axiom. We recommend using the axiom definition for fairly standard groupings that follow straightforwardly from the axioms – commutativity for example – and using separate axiom group definitions for transformation system-related grouping – ordering rewrites in stages, for example. A few suggested axiom groups are:

ac — associative-commutative – a (possibly non-terminating) reordering of an expression

simplify — right-hand side is preferred over left-hand side; repeated application should terminate

propagate — introduction and propagation of properties across expressions

3 Rewriting with Axioms

The jump from having axioms to using them for optimisations isn't far. The proposed C++ standard already suggests that compilers may use axioms for simplifying code, and there already exists a proof-of-concept for axiom-based optimisation [18], based on ConceptGCC. Similar ideas have been used successfully in systems like TAMPR [3] and CodeBoost [1].

3.1 Basic Rewriting

Let's start with a brief explanation of how rewriting works, for those unfamiliar with the concept. A conditional rewrite rule consists of a match pattern, a replacement pattern and a condition. If the condition is always true, it can be omitted. The patterns may have variables (sometimes called meta variables, to distinguish them from C++ variables). For example, consider the following rewrite rule:

$$g(f(x)) \to h(x)$$

$g(f(x))$ is the match pattern, $h(x)$ is the replacement pattern, and x is a variable. If the rule is applied to the expression $g(f(42))$, the result will be $h(42)$. Variables in the replacement pattern should be a subset of the variables in the match pattern and condition.

[6] This plays nicely with C++, but leads to extra verbosity. It may be reasonable to allow *Monoid::Identity* as a shortcut.

We may derive a rewrite rule from any axiom formed from a conditional equation, simply by choosing one side of the equation as the match pattern, and the other as the replacement pattern. If we choose the other way around, we get the inverse rule. In case of a simplification rule, going one way is preferred, but in the case of a commutativity rule, either rewrite direction can be useful. The axiom's parameter list defines the variables of the rule.

Rules are applied to expressions. If we wish to apply a rule to a whole program (i.e., all expressions in the program), we must use a rewriting strategy (discussed below). A rule may either *succeed*, if it matches and its condition is true, or it may *fail*. If it succeeds, the rewrite is performed and we are done. If it fails, we may try other rules if we are applying a group of rules, until we find one which succeeds. A typical rewrite strategy would visit all expressions in a program, and repeatedly apply rules until no rules succeed.

It is important to note that rewriting is not merely syntactic, it also takes into account the types and signatures of the expressions. For example, a rule derived from this axiom

```
axiom Commute(int a, int b) : ac {
  assert(a + b == b + a);
}
```

will apply to an expression $5+4$, but not to $4.2+6.9$, which uses the floating-point plus operator.

Rewrite rules are typically given names, in our system the name follows from the axiom name. Rule names share the same name space as axiom group names. Rules are allowed to have the same name – in that case they form a group and will be applied together (i.e., tried in an arbitrary order until one succeeds or all have failed).

We can expect a rewrite system based on C++ axiom to be both non-confluent (i.e., applying rules in a different order gives a different result) and non-terminating. Rewrite strategies together with axiom groups provide a pragmatic solution to this, and allow us to carefully control the application of rules.

3.2 Rewrite Strategies

We can get pretty far basing an optimisation tool on just one or a few fixed rewriting strategies, and linking each strategy to a particular axiom group. For example, a possible default optimisation strategy would be to do a bottom-up simplification (using the *simplify* group) of the program tree, modulo *ac* rules. Axiom-based optimisation could then be turned on by a compiler option like *-frewrite-rules*.

We can do better, of course. User-defined strategies allow detailed control over when and how various rewrites are applied. Boyle showed in the TAMPR system [3] that such control was needed to obtain many of the goals of program optimisation from rewrite rules.

```
"strategy" Identifier "(" ParamDecls ")" ";" -> StrategyDecl
"strategy" Identifier "(" ParamDecls ")" StrategyBody
                                                -> StrategyDef
"{" Statement* "}"                              -> StrategyBody
```

Fig. 4. Syntax for strategy definitions – this may be parsed as a definition of a function returning a value of type 'strategy'. A limited set of statements and operators (detailed in Fig. 5) for the strategy language usable in the *StrategyBody*.

`true`	do nothing – always succeeds
`false`	do nothing – always fails
`all(`*`s`*`)`	apply *s* to all children of the current node.
`s1 || s2`	apply either *s1* or *s2*, trying *s1* first
`s1 && s2`	apply *s1* then *s2*
`a`	apply rewrite rules name *a*
G	apply any rule from axiom group *G*
G`[`*a*`]`	apply any rule named *a* from axiom group *G*
`repeat(`*`s`*`)`	apply *s* repeatedly until it fails.

Fig. 5. Some suggested strategy combinators and builtin strategies.

Basing ourselves upon an existing program transformation language like Stratego [4], we can make its strategy-building constructs available within C++-like syntax (see Fig. 4), and then either compile our rewrite rules and strategies to Stratego code, or execute them using a Stratego interpreter. The above default strategy may be encoded like this:

```
strategy simple_opt() {
  bottomup(repeat(simplify || (ac && simplify)));
}
```

where *bottomup* does a bottom-up traversal of the program tree, and *repeat* applies is argument until it fails. The choice combinator `||` tries its left argument, then its right argument if the left fails (corresponds to `<+` in Stratego), and the sequence combinator `&&` applies its arguments in sequence and succeeds only if both succeed (corresponds to `;` in Stratego). The two group applications *simplify* and *ac* will apply the actual rewrite rules at the current position in the program tree. See Fig. 5 for an overview of strategy combinators.

Strategies like *simple_opt* may then be made available to the user as a compiler or transformation tool option, e.g. *-frewrite-using=simple_opt*.

3.3 Integrating With Other Optimisations

Often an optimisation rule may only be applied at a particular level of abstraction. Inlining, for example, will commonly expose some opportunities for rule application, while hiding others. For example, in an expression $a + f(b)$

we may be able to inline or do partial evaluation of f and figure out that it returns a zero – thus allowing us to eliminate the + (using a rule derived from *Monoid::Identity*). But if we inline the plus (e.g., in the case of vector addition), our rule would no longer match.

Exposing an inliner interface to the strategy language would be quite useful. In our earlier work on user-defined rules [1] in C++, we added inlining rules for simple functions to open up optimisation opportunities that may otherwise have been lost. For example, we might have a rule

```
axiom GetElement(T a, T::index_type i) : inline {
  assert(a[i] == a.data[int(i)]);
}
```

inlining the code of the user-defined `[]`-operator. Since this is just a simple duplication of the implementation of `[]`, getting the compiler's inliner to do the job would be more general and preferable.

Substitution of expression assignments may also be helpful. For example, consider the rule $a * x + y == \text{axpy}(a, x, y)$ that combines a multiply and an addition into a single operation. We may not be able to tell that the rule can be applied if the operations occur in different statements (possibly far apart in the code):

```
Vector a, b;
a = 5 * a;
b = a + b;
```

But if the expression assigned to a is substituted in the last statement,

```
b = 5 * a + b;
```

we may be able to transform to a more efficient

```
b = axpy(5, a, b);
```

Combined operations like *axpy* are common in numerical libraries like BLAS, when working on large data structures like vectors and matrices. Supplying such rules together with a library saves the programmer from having to remember all the optimised special-case forms – and also means that new optimisations can be added later without having to rewrite existing code.

3.4 Properties and Propagation

Certain axioms may depend on certain properties being fulfilled. For example,

```
axiom SortSort(T a) : simplify {
  if(sorted(a))
    assert(sort(a) == a);
}
```

This axiom can be used to eliminate unnecessary sorting of an already sorted data structure. It is perhaps not very likely that a programmer will ask for

an array to be sorted again just after it was sorted, but by using data-flow analysis, we may track this kind of information throughout the program.

To do this, we need to figure out when an array is sorted (we'll simply call it an 'array', even though it might as well be any ordered data structure). A just-sorted array is sorted:

```
axiom Sorted(T a) : propagate {
  assert(sorted(sort(a)));
}
```

Furthermore, removing an element from an array results in an array that is still sorted (well, at least it does for our kind of array):

```
axiom SortedRemove(T a, T::index_type i) : propagate {
  if(sorted(a))
    assert(sorted(a.remove(i)));
}
```

The *propagate* group is used to identify axioms that may be suitable for data-flow propagation of properties. Any boolean predicate like *sorted* above is usable as a propagated property. Note that without propagation axioms we are unable to assume that any modification (e.g, call to a non-const function) of an object preserves its properties.

Properties may be used as a basis for choosing a more efficient algorithm. For example, the axiom

```
axiom SortedSearch(T a, T::value_type e) : speedup {
  if(sorted(a))
    assert(linsearch(a, e) == binsearch(a, e));
}
```

allows us to choose binary search over linear search of a sorted array. We have chosen the axiom group *speedup* for this axiom, as it's a slightly different concept from expression simplification. Conceivably, our optimisation strategy may put more work into proving that a speedup rule can be applied, than it needs for a simplification rule.

Tracing properties of objects is often useful in numerical programming, where certain operations can be drastically faster if it is know that the operands have special properties, like symmetry in matrices, for example.

4 Axioms for Testing

Once we have axioms and rewrite rules based on them, we'll want to check that our implementation satisfies the axioms. In particular, before we apply optimisation rules to a program, it is a good idea to check that the rules won't change the meaning of the program. While axioms may be used for formal program verification, this is difficult to achieve in a general-purpose language. We can however take a more pragmatic approach, and use the axioms as a

basis for testing.

Using axioms as test oracles[7] is straight-forward – fill in test data for the free variables, and see if the axiom evaluates to true [13,12,7]. It is a pity this kind of specification-based testing isn't made more apparent in the upcoming standard, as it would be a good motivation for actually writing axioms in programs.

Providing basic support for testing is quite simple – we only need to make the instantiated (after concept mapping) axiom code available as callable functions. The testing code may then be called from a test program, or from a testing framework (like JUnit [15] for Java).

For example, referring to the *Identity* axiom for *Monoid*, we may test that it holds for integers, by calling it with a few different integer values:

```
for(int i = -2; i <= 2; i++) {
  Monoid::Identity(std::multiplies<int>(), i);
  Monoid::Identity(std::plus<int>(), i);
}
```

Here, *std::multiplies* and *std::plus* refers to predefined class wrappers for built-in operators. They are necessary because of how the monoid concept is defined (with the operation as a template parameter) – the operator parameter is not really a free variable in the axiom.

4.1 Axioms With Complex Testing Code

Axioms used for testing can be written in any (computable) logic. For testing purposes, it therefore makes sense to allow arbitrary C++ code inside an axiom definition – though this is not allowed in the proposed C++ standard. For instance, we may state that two arrays are equal if they have the same number of elements, and that the elements are equal

```
axiom ArrayEqual(Array a, Array b) {
  bool eq = a.size() == b.size();
  if(eq)
    for(int i=0; i<a.size; ++i)
      eq &= a[i] == b[i];
  assert(eq == (a==b));
}
```

The first lines are needed to iterate through the data set and accumulate information about its components. The *assert* keyword helps to identify which part of this statement sequence is actually the test which defines the axiom. It could be given its C library meaning – abort the program if the test fails – or we could make a more elaborate implementation that counts the number

[7] A *test oracle* is something that tells you the correct result for an operation you want to test.

of failures and successes and records the axioms that fail. It may even be useful to allow *assert* to have additional parameters, e.g., for adding extra information about the test.

Although just allowing simple expressions in axioms is nice, being able to write support code for an axiom provides us with more expressive power. It is also possible to live without "helper" code – then we would need to encapsulate the helper code as a boolean function, possibly making simple axioms more complicated.

Note that while the above axiom may seem trivial, properly testing the implementation of equality is important in order to be sure that other axiom tests relying on it work correctly.

4.2 Testing Exception Behaviour

It is also useful to state as an axiom that a method should throw an exception under specific conditions:

```
axiom DivZeroThrows(T x, T y) {
 if(y.iszero())
   try { div(x, y); assert(false);}
   catch(DivisionByZero) {assert(true);}
}
```

The first assertion tests the lack of an exception being thrown. The second confirms the expected catching of an exception. If helper statements are not allowed, this axiom also needs to be encapsulated.

4.3 Limitations on Testing

The tests derived in this fashion are clearly only as good as the axioms they are based on. If the axioms are wrong, the tests will also be wrong (though one is likely to discover this if the implementation one is testing is correct).

Tests based on the equalities and other comparisons rely on the comparison being correctly implemented. Also, if the two sides of an equality are faulty, but give equal results, this will go unnoticed. In practice, though, comprehensive testing with varied test data and multiple axioms is likely to uncover that something is wrong, even if circumstances conspire against some of the axiom tests.

Effective testing requires good test data. This is something we haven't considered here – we have relied on the programmer supplying appropriate test data. The danger here is that a programmer's assumption of what constitutes good test data is often wrong – the bugs are hiding where one least expects them to be. We will revisit this subject later (Sect. 6).

5 Implementation Issues

The biggest hurdle facing any implementation of a C++ extension is implementing support for the base language. Existing C++ frontends are either far away from supporting the full C++, or are difficult to extend, making the cost of prototyping new language features high.

Support for concepts and axioms has been implemented in the experimental ConceptGCC compiler [9]. At least one attempt to implement axiom-based rewriting (based on ConceptGCC and the C++0x proposal) has been made [18] (and they note that rule application and pattern-matching on the internal GCC representation is indeed quite challenging).

We have not implemented the features described in this paper, but we do have previous experience implementing axiom-based rewriting for C++ [1] and deriving C++ tests from axioms [13]. In this section we will briefly sketch how a prototype may be implemented for C++.

5.1 Translating Axioms to Rules

The syntax used in axioms differs from actual C++ code – people write *a + 0*, not *op(a, identity_element(op))*. This means that to apply rules to user code, we must first translate the rules according to concept map mappings, inlining any concept map functions (like *identity_element*). For example, the *Identity* axiom

```
op(x, identity_element(op)) == x;
```

instantiated for *Vector::plus* and *Vector* becomes,

```
Vector::plus()(x, identity_element(Vector::plus())) == x;
```

then if we inline *Vector::plus()* and *identity_element()*, we get

```
x + Vector::zero == x;
```

which is what we would expect to see in user code.

Overload resolution from the C++ frontend can be used to figure out the signature of operations in the rule (e.g., integer addition as opposed to floating-point addition) – this information is then used in building the match pattern for the rule. Axiom parameters become variables in the match pattern. We used this idea in our previous implementation of user-defined rules.

Note that rewrite rules can be applied to generic template code when the template parameters are constrained by concepts. Without concepts, you'd have to instantiate the template to see which classes are used before you could apply rewriting. Template code making use of concepts will already be written in the same terms as the axioms (e.g., using *op* and *identity_element*), so less work has to be done to adapt the rules.

5.2 Overall Transformation Process

The overall processing of axioms as rules may proceed as follows. First, we need to parse and perform semantic analysis on the program, giving an abstract syntax tree (AST) annotated with type and signature information. The frontend's overload resolution should also be applied to axioms (either now, or after concept mapping) so that the full signature of functions etc. in calls are available for matching in rules.

The axioms and strategy definitions are then picked out from the program tree. Axioms may then be compiled to rules that apply at the concept level, and after applying concept maps, to rules at the class/user level. Testing code is generated after applying concept maps.

Once the rules are available, we apply them according to our built-in or user-defined optimisation strategy. Rule application should be combined with inlining and data-flow analysis for maximum benefit.

6 Discussion

Embedding optimisation rules in programs is not a new idea. User-defined rules in CodeBoost [1] were inspired by the rewrite rules in the Glasgow Haskell Compiler [14]. The CodeBoost implementation was more advanced, however, and supported both conditions and multiple strategies (through a simpler version of the axiom groups introduced in this paper). Conditional rewrite rules is well known from transformation languages such as Stratego [4] and ELAN [2], both of which also support strategies, and from term rewriting in general.

The CodeBoost user-defined rules were limited to rewriting, and although we derived optimisation rules from a formal algebraic specification, the actual coding of the rules was done by hand and related directly to C++ classes. Concepts in C++ let us bridge the gap between the specification (written in terms of algebraic ideas like monoids, fields, rings etc.) and the concrete implementation as C++ classes. Writing the relevant parts of our specifications in concept syntax, with axioms, will make the axioms automatically available for use by tools. There is still a piece missing in the puzzle, though – tying the information from concepts and axioms together with some form of structured documentation. No doubt someone is already working on this problem.

Support for axiom-based rewriting for ConceptGCC [18] follows the C++ standard proposal more closely than we do here. Rule application in [18] is for now restricted to contexts explicitly constrained by concepts (see Sect. 2), and transformation is restricted to a single strategy (leftmost-outermost reduction). The goal is to have a framework for concept based optimisation that may eventually replace type-specific built-in simplifications in the compiler. We do not aim to compete with this implementation, though we hope that some of our ideas will be useful for their work.

Axiom-based rules as described here is limited to operating on expressions, and will not be fully effective if the transformation system isn't able to combine expressions from multiple statements (e.g., nesting expressions as much as possible). There is a trade off here between rule-based optimisations, and optimisations like common subexpression elimination – duplicating some common subexpressions may open up for rule applications but can also lead to duplicated work.

Our initial experiments with rules had quite promising results. A small number of rules from the specification of the Sophus numerical library was used (together with general simplification rules) to optimise the C++ implementation of Sophus, giving 5–10 times speedup (the latter after the source code was simplified to take advantage of the rules) as well as reduced memory use. Some of the speedup was due to other optimisations in the system – running without rule-based optimisations gave up to 2 times speedup.

Applying axiom-based rules to existing program code may not give quite as good results, since programmers often have done many of the same optimisations by hand already. We expect the full benefit to be more apparent with previously unoptimised programs written in a high-level style, and when delivered together with a performance sensitive library, such as for numerical software – an idea also known as *active libraries* [6].

Using axioms as a basis for testing is known from systems like DAISTS [7]. Compared to unit-testing frameworks like JUnit [15] – and the xUnit family of frameworks for other languages – the advantage of axioms is the separation of test code from test data. This means that one may easily test one axiom with many different data values, and use the same values to test many different axioms. Comprehensive testing can be done by building libraries of axioms and test data, and testing all axioms against all suitable data. *Theories*, available in JUnit 4.4 allow universal quantification [17], and may be called from testing code like axioms in this paper. JUnit 4.4 also allows automatic application of available test data to tests.

QuickCheck [5] is a testing system for Haskell where a programmer can state *laws* (like C++ axioms) as Haskell functions. The *quickcheck* testing function generates random test values, and tests that a law holds. Conditional laws are also allowed – in that case more test values will be generated as necessary to test the law sufficiently. The algebraic data types in Haskell makes generating random data structures fairly simple, and the programmer can supply generator functions to fine-tune the data generation. There is no link between QuickCheck and rewrite rules in the Glasgow Haskell Compiler (mentioned above).

Partition testing is another testing approach where one attempts to divide the input domain into regions and test only one value from each region (and some boundary values). This does not necessarily give better results than random testing, though [11]. Both partitioning and random data generation will likely require some help from the C++ programmer. This is an area that

should be explored further in order to make full use of axiom-based testing.

6.1 Conclusion

Concepts and axioms, which are being introduced in the upcoming C++0x standard, are language features that may prove quite useful for both documentation, automated testing and program optimisation. To reap the full benefit of axioms, we have introduced a few additional language features:

- *Axiom groups* for classifying axioms according to how they may be used
- *Strategies* for specifying how and when axiom-based rules should be used
- *Callable axioms / axiom groups* to make axioms usable for testing
- *Properties* which may be propagated according to axioms, and used in axiom conditions

Allowing a wider range of statements in axiom bodies, and allowing axioms to occur outside concepts, gives us some added usability benefits over the proposed standard.

Our main contribution over our own and others' previous work in this area, is setting it in the context of the upcoming C++ standard, and tying together the ideas of specification-based optimisation and specification-based testing.

Acknowledgements: Thanks to Valentin David for useful comments and help with the intricacies of C++ and grammars, and thanks to the referees for many useful comments and tips.

References

[1] Bagge, O. S. and M. Haveraaen, *Domain-specific optimisation with user-defined rules in CodeBoost*, in: J.-L. Giavitto and P.-E. Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03)*, Electronic Notes in Theoretical Computer Science **86/2** (2003).

[2] Borovanský, P., C. Kirchner, H. Kirchner, P.-E. Moreau and C. Ringeissen, *An overview of ELAN*, in: C. and H. Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications* (1998).

[3] Boyle, J. M., T. Harmer and V. Winter, *The TAMPR program transformation system: Simplifying the development of numerical software*, in: E. Arge, A. M. Bruaset and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, Birkhäuser, Boston, 1997 pp. 353–372.

[4] Bravenboer, M., K. T. Kalleberg, R. Vermaas and E. Visser, *Stratego/XT 0.16: Components for transformation systems*, in: F. Tip and J. Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation* (2006).

[5] Claessen, K. and J. Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, in: *ICFP '00: Proceedings of the fifth ACM SIGPLAN*

international conference on Functional programming (2000), pp. 268–279.

[6] Czarnecki, K., U. W. Eisenecker, R. Glück, D. Vandevoorde and T. L. Veldhuizen, *Generative programming and active libraries*, in: *Selected Papers from the International Seminar on Generic Programming* (2000), pp. 25–39.

[7] Gannon, J., P. McMullin and R. Hamlet, *Data abstraction, implementation, specification, and testing*, ACM Trans. Program. Lang. Syst. **3** (1981), pp. 211–223.

[8] Gottschling, P., *Fundamental algebraic concepts in concept-enabled C++*, Technical Report TR639, Department of Computer Science, Indiana University (2006).

[9] Gregor, D., J. Järvi, J. Siek, B. Stroustrup, G. D. Reis and A. Lumsdaine, *Concepts: linguistic support for generic programming in C++*, in: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2006), pp. 291–310.

[10] Gregor, D., B. Stroustrup, J. Siek and J. Widman, *Proposed wording for concepts (revision 3)*, Technical Report N2421=07-0281, JTC1/SC22/WG21 – The C++ Standards Committee (2007), http://www.open-std.org/jtc1/sc22/wg21/.

[11] Hamlet, D. and R. Taylor, *Partition testing does not inspire confidence*, IEEE Trans. Softw. Eng. **16** (1990), pp. 1402–1411.

[12] Haveraaen, M., *Institutions, property-aware programming and testing*, in: *Proceedings of the ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)* (2007), pp. 23–34.

[13] Haveraaen, M. and E. Brkic, *Structured testing in Sophus*, in: *Norsk Informatikkonferanse NIK 2005* (2005), pp. 43–54.

[14] Jones, S. P., A. Tolmach and T. Hoare, *Playing by the rules: Rewriting as a practical optimisation technique in GHC*, in: R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, 2001.

[15] Louridas, P., *JUnit: Unit testing and coding in tandem*, IEEE Softw. **22** (2005), pp. 12–15.

[16] Mosses, P. D., editor, "CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language," Lecture Notes in Computer Science **2960**, Springer-Verlag, 2004.

[17] Saff, D., *Theory-infected: or how I learned to stop worrying and love universal quantification*, in: *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion* (2007), pp. 846–847.

[18] Tang, X. and J. Järvi, *Concept-based optimization*, in: *Proceedings of the ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)* (2007).