User-Extensible Simplification—Type-Based Optimizer Generators

Sibylle Schupp[1], Douglas Gregor[1], David Musser[1], and Shin-Ming Liu[2]

[1] Dept. of Computer Science, RPI {schupp,gregod,musser}@cs.rpi.edu
[2] Hewlett Packard shin@cup.hp.com

Abstract. For abstract data types (ADTs) there are many potential optimizations of code that current compilers are unable to perform. These optimizations either depend on the functional specification of the computational task performed through an ADT or on the semantics of the objects defined. In either case the abstract properties on which optimizations would have to be based cannot be automatically inferred by the compiler. In this paper our aim is to address this level-of-abstraction barrier by showing how a compiler can be organized so that it can make use of semantic information about an ADT at its natural abstract level, before type lowering, inlining, or other traditional compiler steps obliterate the chance. We present an extended case study of one component of a C++ compiler, the simplifier; discuss the design decisions of a new simplifier (simplifier generator) and its implementation in C++; and give performance measurements. The new simplifier is connected to the Gnu C++ compiler and currently performs optimizations at very high level in the front end. When tested with the Matrix Template Library, a library already highly fine-tuned by hand, we achieved run-time improvements of up to six percent.

1 Introduction

The introduction of abstract data types (ADTs) marks an important step for software production. By encapsulating implementation details of a type, abstract data types allow new separations of concerns and play a large role in making code sharing and large scale programming more feasible, especially after the extension of ADTs in the 1980s and 1990s to the object-oriented programming and generic programming styles. Although data abstraction is accepted in theory without reservation, it is fair to say that there is still considerable hesitation when it comes to using this style in practice. Ironically, the larger and more computationally intensive a project is, and the more it could take advantage of the safety and robustness an object-oriented approach provides, the stronger the opposition. The main concern articulated is efficiency, based on widespread experience that non-object-oriented programs outperform object-oriented ones, even when, in hybrid languages, features like dynamic method binding or inheritance are used cautiously. Depending on the programming language used, some simply accept overhead, while others resort to a programming style that anticipates and

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 86–101, 2001.

then avoids the overhead of object-based programming. Thereby, however, they frequently sacrifice coding styles considerably. Especially in C++ there now exist high-performance libraries, e.g., BLITZ [33], the Parallel Object Oriented Methods and Applications project (POOMA) [24], and the Matrix Template Library (MTL) [28], that are able to compete with the efficiency of Fortran code, but pay for the performance gains with cryptic code at the user's level. For example, a simple operator expression in these libraries is written as a function call with additional parameters or, even more unexpectedly from the user's view, by a so-called expression template, a nest of class templates—constructs that allow library designers to hand-optimize away temporary variables and to thus overcome the "abstraction penalty."

In this paper we address this level-of-abstraction barrier in a different way. We suggest that optimizations of ADTs take place during compilation and not in the source code. At the same time we suggest performing these optimizations directly at the level of ADTs. We claim that ADTs offer opportunities that are not available at a lower level of abstraction and would be obliterated by premature type lowering, inlining, or other traditional compiler steps. Since these optimizations either depend on the functional specification of the computational task performed through an ADT or on the semantics of the objects defined, and cannot be automatically inferred by a compiler, our approach is characterized by a flow of semantic information from the user program to the optimizer. While traditionally a compiler is a program that came into existence long before the user program, is applied to it as a black box, and delivers an executable without any communication with the program designer, a compiler in our understanding rather is a compilation generator that depends on (static) user information for its completion. With users extending the semantic knowledge available to the compiler, it can operate on strong assumptions about a type's behavior and, as a result, generate more efficient code. Especially library designers, therefore, might find our approach useful: once they have provided the semantic information of a data type, its optimizations are available not only in all further compilations or linkings but also to all applications on top of the respective type. Thus, library clients get the benefit of efficient abstract data types in an entirely transparent way. When we use the term "user" in the following, we therefore mostly envision library designers and the like.

For the rest of the paper we focus on C++ [31]. Restricting ourselves to one component of a C++ optimizer, the simplification component, hence to the optimization of expressions, we show how to organize this component so that a normal C++ program can provide the semantic information needed to statically simplify an ADT in a correct and optimal way. In section 2 we summarize both the process of traditional simplification and the specification of the new simplifier. Key in decoupling the process of simplification and the semantics of data types is the *concept-based* approach, an approach that is increasingly used in the design of reusable software components, but also in compiler design in the related sense of *formal concept analysis*. In section 3 we briefly characterize programming with concepts, explain its advantages for the design of generative

optimizers, and illustrate concept-based simplification rules. We summarize the major design decisions behind the implementation in section 4 and illustrate our approach using an extended example of an algorithm of the MTL that is instantiated with a LiDIA [32] data type. In section 5 we demonstrate how the LiDIA user can guide the simplifier to replace expressions used in the MTL by more efficient LiDIA expressions—without changing the MTL code or compromising the readability of the source code. The statistical data of the run-time and code-size improvement of this and other experiments are reported in section 6. The approach of generative compilation seems to fit well in a recent trend of designing compilers that are more flexible or responsive. We compare our approach to others in more detail in section 7 and conclude the paper with a brief discussion of our future plans.

2 Simplification

We selected the simplification component because of its minimal dependencies on other components. In particular we wanted to avoid the need for any modifications to the data dependency analysis part of an optimizer. We will see, however, that the new approach to simplification not only extends the original functionality to user-defined types, but opens up new kinds of high-level optimizations. In the next subsections we summarize both traditional simplification and the features of our proposed generalization, including its benefits specifically for high-level optimizations.

2.1 Algebraic Simplification

A simplifier is an optimization component that operates on expressions. It rewrites a given expression (expression tree) to a different expression which is in some sense simpler: either as an expression tree of lower height or as an expression with cheaper operators, or as a combination of both. Its semantic knowledge base is a set of simplification rules that are subsequently matched against the expression under consideration; if no simplification rules applies, the expression is returned unchanged. A standard example of a simplification rule is

$$x + 0 \to x. \qquad (*)$$

When this rule is applied, it replaces the addition by its first argument and thus saves carrying out the addition. Simplification rules can apply to more than one operator as in

$$(x\,!=\,0)\mid(y\,!=\,0)\to(x\mid\,y)\,!=\,0$$

and furthermore can be conditional:

$$\max(x,c)=x \quad \text{if } c \text{ is the smallest possible value.}$$

Since constant arguments like the 0 in the first simplification rule (*) rarely appear directly in user code, simplification traditionally takes place after other

parts of the optimizer have transformed a user-defined expression, possibly several times and from several perspectives. In the SGI MIPS and Pro64 compilers, for example, constant folding, constant propagation, array analysis, loop fusion, and several other loop transformations all have a subsequent simplification step that cleans up the previous results. Whether high- or low-level, however, simplification rules are phrased in terms of integral or floating-point expressions and, with a few exceptions of complex expressions, do not even apply to the classes of the C++ standard library, such as `string`, `bitmap`, `bool`, or the types of the Standard Template Library, much less to user-defined types.

2.2 User-Extensible Simplification

What are the characteristics of the proposed simplifier? First, it can be extended by arbitrary user-defined types. Admitting arbitrary types, however, gives rise to the question whether the current simplification rules and their operators, designed mainly for integers and floating-point numbers, are "substantial enough" to represent frequently occurring user-defined expressions that are worthwhile to optimize. Quite certainly, a numeric library uses different expressions with different frequencies than, say, a graphics package. If we admit new types but keep the simplification rules the same, the benefits of simplification might be severely restricted. Therefore, users should have the opportunity to extend the simplifier by their own operator (function) expressions as well as by simplification rules for those.

Next, all extensions should be easy to do. Although we can expect that users who are interested in the subtleties of performance issues are not programmer novices—as already mentioned, we envision library designers and the like as our special interest group—we certainly do not want them to have to modify the compiler source code. All extensions, in other words, should be done at the user's level, through a user's program. In section 3 we will explain how a concept-based approach helps to realize this requirement; here we only point out that all user-defined extensions are organized in header files and are compiled along with the main program. In the current implementation users have to manually include two header files—the predefined header `simplify.h` and the file with their own, application-specific simplifications—but we will automate the inclusion. All that's left to do, then, is to turn on the simplifier with the optimization flag `-fsimplify`:

```
g++ -O3 -fsimplify prog.C
```

In summary, the new simplifier is characterized by the following properties:

- It is extensible by user-defined types, operators and function identifiers.
- It is extensible by user-defined simplification rules.
- All extensions are specified in ordinary user programs.
- It is organized so that the extensions do not entail any run-time overhead but are processed entirely at compile time.

Given a simplifier with these capabilities, users can then in fact go beyond the traditional, algebraic nature of simplification rules and use the simplification process to rewrite expressions in non-standardized ways. Thus far, we have come across two motivations for the introduction of non-algebraic rules: one is the separation of readability and efficiency at the source code level and the other is the possibility of rewriting certain expressions for the purpose of invoking specialized functions. In the first case simplification rules can be used to perform optimizations that otherwise take place in the source, e.g., of high-performance libraries. Rather than replacing operator expressions manually and at the user's level by expression templates or other constructs that are efficient but hard to digest, library designers can introduce simplification rules that automatically rewrite expressions in the most optimal form. Thereby, neither readability nor efficiency is compromised. The other case occurs in the situation of parameterized libraries, where one library is instantiated with types of another, providing optimization opportunities the carrier library did not anticipate. Without the need to change this carrier's code, simplification rules can be used here for replacing one function invocation by another. In both cases, the extensible simplifier allows library designers to pass optimizations on to the compiler that otherwise have to be done manually and directly at the source code level; for examples of both sets of rules see section 5.

3 Concept-Based Rules

Simplifying user-defined types, at first glance, might look like an impossible task because the types to be optimized are not known at simplifier design time. Without any knowledge of a type, its behavior or even its name, how could the simplifier treat it correctly and efficiently? However, the situation is similar to the situation in component-based programming where the clients and servers of a component also are unknown at component design-time—and yet the interaction with them has to be organized. In component-based programming there are several techniques available to model the necessary indirection between a component and its clients. The one that seems most appropriate for optimizer generators is the *concept-based* approach.

There are many definitions of "concept," but as we use the term here, it means a set A of *abstractions* together with a set R of *requirements*, such that an abstraction is included in A if and only if it satisfies all of the requirements in R. This definition is derived in part from formal concept analysis [35,36], in which, however, the abstractions are usually merely names or simple descriptions of objects. Another source is the Tecton concept description language ([13, 14]; see also [21]), in which the abstractions are more complex: they are algebras or other similar abstract objects such as abstract data types. An informal but widely known example of this understanding of concepts is the documentation of the C++ Standard Template Library developed at SGI [2]. There, the abstractions are types, and concepts like Assignable or Less Than Comparable are used to explicitly specify which requirements each type must satisfy in order to be

used as a specialization of a certain template parameter. In analogy, we state a simplification rule at a conceptual level, in terms of its logical or algebraic properties. While traditional simplification rules are directly tied to the types to which they apply, the concept-based approach allows referring, for example, to the rule (*) of section 2 abstractly as Right-identity simplification (fig. 1 lists more examples; for the representation of concepts in C++, see [26]).

left-id op X	$\rightarrow X$	$X \in M$	(M, op) monoid
X op right-inv(X)	$\rightarrow$ right-id	$X \in G$	(G, op) group
$X = Y \cdot Z$	$\rightarrow$ multiply(X,Y,Z)	X, Y, Z LiDIA::bigfloat	
X = bigfloat(0)	$\rightarrow X$.assign_zero()	X LiDIA::bigfloat	

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

What are the advantages of the concept-based approach? Most importantly, concept-based rules allow for the decoupling of rules and types (type descriptors) that makes the extension of the simplifier possible. Since only requirements are mentioned and no particular types, the scope of a simplification is not limited to whichever finite set of types can be identified at simplifier design time. Instead, a rule can be applied to any type that meets its requirements. At the same time, the simplification code can be laid out at an abstract level and independently from particular types; it can contain the tests that check for the applicability of a rule and can symbolically perform the simplification. Secondly, concept-based rules help to reduce the amount of code of the compiler. The abstract Right-identity simplification rule, for example, corresponds to 16 concrete instances in the Pro64 SGI compiler, including $x + 0 \rightarrow x$, $x \cdot 1 \rightarrow x$, $x\&\&1 \rightarrow x$, or $\max(x, c) = x$ if c is the smallest possible value. Lifting several concrete rules to one abstract rule lowers the maintenance and debugging costs of the compiler itself. Lastly, abstractions makes user-extensions manageable. If users describe their types in the same abstract way that the simplifier states each of its rules, type descriptors can be provided independently from any particular rule and without any knowledge of the current set of simplifications. An internal query system can then check for superconcept/subconcept relations between the requirements of a rule and the behavior of a given type. The next section further discusses the implementation.

4 Implementation

The simplification framework is divided into three parts: the core simplifier or simplifier generator, which is the major part; an interface from a particular C++ compiler to the core simplifier; and the set of user-defined extensions (possibly empty). The core simplifier is currently connected to the Gnu compiler front end and can be used as-is with the Gnu and the Pro64 compilers, and others that use the Gnu front end. It is, however, a stand-alone program, which, if an appropriate

interface from the internal representation of the previous compilation step to the simplifier's internal representation is provided, can work with any other C++ compiler that is standard-compliant and supports in particular all advanced template features.

Of particular importance for our purpose is the C++ feature of specialization, including partial specialization, where a set of class templates specialize in different ways one common class template (primary template, in C++ jargon). Originally, specialization was introduced to support different levels of genericity and different implementations of a generic task. The main advantage in our situation, however, is the fact that the compiler selects the most appropriate template based on the specializing type. Partial specialization is furthermore responsible for the Turing-completeness of the template sublanguage of C++—branching is modeled through template specialization, iteration through recursive templates—which allows us at least theoretically to implement simplification as a static program.

The key idea of the simplifier generator is in fact to express the generator entirely within the template sublanguage. Provided simplification rules are correctly represented as class templates, the normal instantiation mechanism of C++ and the partial order of partially specializing templates can then be used not only to select the best fitting simplification rule without additional effort on our part, but also to perform the selection at compile time. In addition, representing each rule as its own class, separated from all others, allows users to add new rules without modifying the existing ones. For this to work, however, we have to reorganize the formal side of a simplification rule as well as its actual instance. On the formal side, it is necessary that the selection of a simplification rule becomes statically decidable. While non-static simplification works on expressions directly, they now have to be modeled as types, so-called *expression templates*. With expressions as types, simplification rules can be represented as class templates so that the left-hand side of a rule constitutes the template interface and the right-hand side the template body. On the actual side of simplification expressions, consequently, we have to provide mappings between the representations that expressions have inside and outside the simplifier generator. Thus, the interface between the simplifier and the Gnu compiler essentially is concerned with the conversion from the Gnu's internal TREE representation [4] to the expression-template representation used within the simplifier. Fig. 2 illustrates the template-based representation of a simplification rule using right-identity simplification. In this example, the class template is a partial specialization of the primary template `Simplify` by two parameters: an abstract binary expression, represented as expression template `BinaryExpr<BinaryOpClass,LeftOperand,RightOperand>`, and the set of constraints that a right-identity simplification imposes on the bindings of the three template parameters (`BinaryOpClass`, `LeftOperand`, `RightOperand`), encapsulated in the type `RightIdentitySimpl`. Together, the two parameters ensure that the simplification class is instantiated only when the bindings of all its parameters meet the requirements of a right-identity simplification. If they are met,

however, the class gets instantiated and the result of the simplification, that is, the right-hand side of the simplification rule, becomes accessible through the type definition in the body of the class. For the binding `BinaryExpr<Add,X,0>`, for example, (the type counterpart to the expression $x+0$) the type `Simplify::result` resolves to `X`, as expected.

```
template<class BinaryOpClass, class LeftOperand, class RightOperand>
struct Simplify<Expr<BinaryExpr<BinaryOpClass,LeftOperand,RightOperand>>,
                                  RightIdentitySimp>
{
  typedef typename Simplify<LeftOperand>::result result;
};
```

Fig. 2. Template representation of the right-identity simplification rule.

To summarize the technical details, the simplifier generator is based on advanced features of generic programming in C++, most notably the already mentioned expression templates [34,12], template metaprogramming [5], and traits (interface templates) [23]. As already mentioned, these features are increasingly used in C++ libraries, but can (or should, as we argued) be moved from the user level to the compiler level.

The set of concept-based rules is complemented by a set of conceptual type descriptors for each type that is the subject of simplification. This is the part where the simplifier is controlled by its users: they are expected to provide information about the logical, algebraic, or computational behavior of their types—whatever properties they omit, the simplifier will not be able to take advantage of. It might seem strange to hold users responsible for defining a type's behavior but in some sense they only extend the responsibilities they already assume in traditional variable declarations. There, they assert their variable to be of a certain type; here, they additionally assert semantic properties. It also might appear that users have to provide a great amount of information or might even have to revise their type descriptors each time the set of simplification rules is extended. If stated abstractly enough, however, only a little information is necessary to integrate a user-defined type. For example, the left column in fig. 3 uses the LiDIA type `bigfloat` to show the complete, slightly idealized code to register this type with the simplifier. Extending the simplifier by new functions or expressions follows a similar scheme: each property the simplifier cannot infer has to be specified. Like type descriptors, the descriptors of functional expressions are standardized and implemented as so-called interface templates (or traits). The right column in fig. 3 gives two examples of traits defining the LiDIA member functions `is_zero` and `assign_zero`. The complete code of all LiDIA-specific extensions can be found in [10].

Concept-based rules and concept-based type (or function) descriptors, finally, are brought together in an internal validation step that checks for each simplification rule whether its semantic constraints are met. Given an actual expression

```
// Algebraic behavior
template<>
struct AlgebraTraits<bigfloat>
{
  typedef Field<bigfloat,
              Add, Mult,0
              UnaryMinus, Sub, 1,
              Recip, Div>
              structure;
};


// Computational behavior
template<>
struct TypeTraits<bigfloat>
{
  typedef __true_type is_applicative;
  typedef __true_type is_floating;
};


// Literal table
template<> bigfloat
LiteralTable<bigfloat>::literals[] =
{
  bigfloat(0),
  bigfloat(1)
};
```

```
struct UnaryOpTraits<
  LiDIA::EqualsZero, bigfloat>
{
  typedef __true_type  is_applicative;
  typedef __false_type has_side_effects;
  typedef __true_type  operand_is_const;
  typedef __false_type can_overflow;
  typedef __false_type can_underflow;
  typedef __false_type can_zero_divide;
  typedef bool         result_type;
  static inline bool
  apply(const bigfloat& operand)
  {
    return operand.is_zero();
  }
};
struct UnaryOpTraits<
  LiDIA::AssignZero, bigfloat>
{
  typedef __false_type is_applicative;
  typedef __true_type  has_side_effects;
  // ...
  static inline void
  apply(bigfloat& operand)
  {
  return operand.assign_zero();
  }
};
```

Fig. 3. Left: user-provided descriptors of LiDIA's type `bigfloat` (complete, but slightly simplified). Right: user-defined expression descriptors of the LiDIA functions `is_zero` and `assign_zero`.

with an actual type the validator retrieves its type descriptor and compares the type specification against the requirements of the corresponding variable of the simplification rule that best fits the given expression. The comparison of requirements could be as simple as a check for equality, but often implies in practice a more complicated confirmation of one concept as a subconcept of another. To derive such relations the validator then searches a small internal repository. Suppose, for example, the rule (*) for right-identity simplification is about to be applied to an instance of, say, LiDIA's `bigfloat` type. The validator first retrieves the requirements of the rule (which are specified for elements of monoid structures) and the description of the type (which forms a field), then performs a series of lookups in the internal repository to confirm the concept of a field as a subconcept of the monoid concept, and finally gives way to the instantiation of the simplification rule with the `bigfloat`-expression; as we have seen earlier

(fig. 2) the body of this class then holds the result of the symbolically simplified expression.

5 Extended Example

To demonstrate the optimization opportunities that an extensible simplifier provides, we now discuss an extended example using the libraries MTL (Matrix Template Library) [28] and LiDIA [32], a library for computational number theory.

```
T a = a_in, b = b_in;
if (b == T(0)) {
  c_ = T(1);
  s_ = T(0);
  r_ = a;
} else if (a == T(0)) {
  c_ = T(0);
  s_ = sign(b);
  r_ = b;
} else {
  // cs=|a| / sqrt(|a|^2+|b|^2)
  // sn=sign(a)·b / sqrt(|a|^2+|b|^2)
  T abs_a = MTL_ABS(a);
  T abs_b = MTL_ABS(b);
  if (abs_a > abs_b) {
    // 1/cs = sqrt( 1+|b|^2/|a|^2 )
    T t = abs_b / abs_a;
    T tt = sqrt(T(1) + t * t);
    c_ = T(1) / tt;
    s_ = t * c_;
    r_ = a * tt;
  } else {
    // 1/sn=sign(a)· sqrt(1+|a|^2/|b|^2)
    T t = abs_a / abs_b;
    T tt = sqrt(T(1) + t * t);
    s_ = sign(a) / tt;
    c_ = t * s_;
    r_ = b * tt;
  }
}
```

```
T a = a_in, b = b_in;
if (b.is_zero()) {                    (1)
  c.assign_one();                     (1)
  s.assign_zero();                    (1)
  r_ = a;
} else if (a.is_zero()) {             (1)
  c.assign_zero();                    (1)
  s_ = sign(b);
  r_ = b;
} else {
  // (see left column)
  //
  T abs_a = MTL_ABS(a);
  T abs_b = MTL_ABS(b);
  if (abs_a > abs_b) {
    // (see left column)
    T t = abs_b / abs_a;
    T tt = sqrt(T(1)+square(t));      (*)
    inverse(c_,tt);                   (2)
    multiply(s_,t,c_);                (1)
    multiply(r_,a,tt);                (1)
  } else {
    // (see left column)
    T t = abs_a / abs_b;
    T tt = sqrt(T(1)+square(t));      (*)
    divide(s_,sign(a),tt);            (1)
    multiply(c_,t,s_);                (1)
    multiply(r_,b,tt);                (1)
  }
}
```

Fig. 4. The body of the MTL algorithm `givens_rotation` with `T` bound to the LiDIA `bigfloat` type: the original MTL code (left) and the code generated by the simplifier (right). Numbers in parentheses at the end of a line indicate the number of saved temporary variables of type `bigfloat`; (*) marks rewrites by more efficient functions.

In short, MTL provides the functionality of a linear algebra package, but also serves as basis for sophisticated iterative solvers. It provides extensive sup-

port for its major data types, vectors and matrices, but relies on C++ built-in types for the element type of a matrix. LiDIA, on the other hand, provides a collection of various multi-precision types, mainly for arithmetic (e.g., in number fields) and cryptography. For the example we selected the MTL algorithm `givens_rotation`, performing the QR decomposition of the same name, and instantiated it with vectors over the LiDIA type `bigfloat`. The core of the function `givens_rotation` is listed in fig. 5. As the code listing (left column) shows there are several operator expressions that, when applied to instances of class types, require constructor calls, hence temporary variables. At the same time LiDIA's `bigfloat` class has member functions defined that are equivalent to, but more efficient than some of these operator expressions: the application of the member function `is_zero`, for example, requires one temporary less than the equivalent operator expression `b == T(0)` (first line) and, likewise, the call `multiply(r_,b,tt)` is more efficient than `r_= b*tt` (last line). In a traditional compilation model, LiDIA users would have to either accept performance losses or would have to modify MTL code; with the extensible simplifier, however, they can leave the MTL source code unchanged, but "overwrite" the generated code.

Using the techniques described in the last section they can register the `bigfloat` type and then specify which operator expressions they want to see replaced. We showed in fig. 3 how to instruct the simplifier so that it replaces test-for-zero expressions on `bigfloat` variables (`EqualsZero`) by calls to their member function `is_zero`. Following this scheme, we added altogether 6 LiDIA-specific rules for the example. These 6 rules, along with the general-purpose rules already in the simplifier, result in the saving of 12 temporaries (see fig. 5, right column); more precisely, the saving of constructing and destructing 12 `bigfloat` instances. The next section reports on the speed-up and code size saving gained.

6 Experimental Results

We conducted a series of experiments where we instantiated different MTL algorithms with LiDIA data types. For the presentation of the results in this section we selected three MTL algorithms in addition to the already discussed Givens rotation: the Householder QR transformation `generate_householder`, `__tri_solve` for solving triangular equation systems, and `__major_norm` for computing the 1- and ∞-norms. All four routines are core routines for linear algebra and numerical computations. Another, more practical reason for selecting them was their code size in the MTL implementation, which, between 25 and 60 lines, seems to be large enough to allow for noticeable simplification effects. In the tests the simplifier was extended by the LiDIA-specific rules listed in figure 5.

MTL provides small demonstration programs for each algorithm that just generate its input arguments, run the algorithm, and print the result. We used these programs, but inserted cycle counters around the algorithm invocation. For the compilation we extended the Gnu compiler g++ 2.96 by an interface file to the simplifier. The version 2.96 of g++ has a well-known, serious performance bug in its inliner, which results in smaller performance improvements in three

$X = \text{bigfloat}(0) \rightarrow X.\text{assign\_zero}()$	$X == \text{bigfloat}(0) \rightarrow X.\text{is\_zero}()$
$X = \text{bigfloat}(1) \rightarrow X.\text{assign\_one}()$	$X == \text{bigfloat}(1) \rightarrow X.\text{is\_one}()$
$X * X \rightarrow \text{square}(X)$	$X = X \text{ op } Y \rightarrow X \text{ op}= Y$, op $\in \{+,*\}$
$X = \text{op } Y \rightarrow \text{op}'(X, Y)$, op $\in \{1/\_, \text{square}\}$, op$'$ $\in \{\text{invert, square}\}$	
$X = Y \text{ op } Z \rightarrow \text{op}'(X, Y, Z)$, op $\in \{+, -, *, /\}$, op$'$ $\in \{\text{add, subtract, multiply, divide}\}$	

Fig. 5. LiDIA-specific simplification rules

cases; in the `__tri_solve` test, we had to manually perform expression rewriting due to problems with the inliner interface, thus `__tri_solve` did not suffer from the same abstraction penalty losses as the other three cases. We then compiled each algorithm with and without the compilation flag `-fsimplify` at the highest optimization level,`-O3`, measured the size of the generated code and counted the cycles at run time. We also varied the precision of the floating point type, but this did not change a program's behavior. To determine the number of cycles we ran each example 100,000 times, eliminated values that were falsified by interrupts, and computed the arithmetical means of the remaining values; each test was repeated several times. The tests were performed on a Linux platform with an AMD Duron processor, using MTL 2.1.2-19 and LiDIA 2.0.1.

The results show that we were able to achieve performance gains in all four cases, substantial gains for the Givens rotation (6%) and the `__tri_solve` algorithm (8%), and still 1% and almost 4% speed-ups in the `__major_norm` algorithm and the Householder transformation. It should be emphasized again both that manual rewriting changes the test conditions for `__tri_solve` and that the problems with the inliner of g++ are temporary and will be solved in its next release. With a fully working inliner without abstraction penalties we expect to further improve on the current performance gains. The speed-ups came along with a slight reduction of the code size in three cases, but the code size went up for the `major_norm` (by 1320 bytes). We suspect, again, the problem was due to the inliner.

Table 1 summarizes the changes to the code size and run time of each algorithm. It also lists for each algorithm its length in lines of code, the number of temporaries of type `bigfloat` that were saved, and, in parentheses, the total number of temporaries that could be saved with our approach but would have required extending the set of LiDIA-specific simplification rules.

7 Related Work

Our approach borrows from programming methodology and our motivation fits in a modern understanding of compilation; in this section we discuss related work in more detail. Programming with concepts, first, is an old approach underlying libraries in several languages, most notably Ada [22] and, with the most success, C++, where the Standard Template Library became part of the language specification [30,20]. The demand for adaptable components, at the same time,

Table 1. MTL algorithms simplified with LiDIA-specific simplification rules

Algorithm	Length (lines)	Temp. saved	Code size (bytes) before	after	savings	Run time (cyles) before	after	speedup
major_norm	28	3(5)	682216	683540	-1324	25704	25523	1.01 %
generate_householder	38	9	688037	687897	140	109343	105335	3.81 %
tri_solve	46	6	688775	688300	475	48095	44434	[8.24%]
givens_rotation	54	12	678450	678037	413	30112	28290	6.44 %

describes a recent trend in software development, which comes in several varieties. Although we were mostly inspired by generic programming in the sense of STL and its successors, the methodologies of adaptive and aspect-oriented programming [16] and intentional programming [29] bear resemblance to our goals but typically work neither with C++ nor with traditional (compilation) environments.

Extensible compilers are another recent trend, along with performance tuning for selected types and the introduction of features that allow advanced users to improve the optimizations their compilers perform. The ROSE source-to-source preprocessor [6], for example, extends C++ by a small generator for optimization specifications for class arrays. In the same direction, but not restricted to arrays, are the "annotation language for optimizing libraries" [11] and the OpenC++ project, part of the Open Implementation project at Xerox [3]. While the former can direct the Broadway compiler, the latter translates meta-objects in a C++ extension to source code that any C++ compiler can process. Particularly in C++, as already mentioned, much work is also done using the language itself, through traits, expression templates, and template metaprogramming, either to enforce certain (statement-level or cache) optimizations or to hand-code them [34,23,5, 27]; see also [9] for a critical evaluation. Both ideas of modifying (extending) the translated language and modifying (customizing) the translated data types, however, are different from our approach of directly getting the optimization to work with arbitrary user-defined types. In performing type-based alias analysis, the latest implementation of the Gnu compiler realizes the advantages of higher programming constructs and type information for optimization tasks, but restricts type-based alias analysis to built-in types [18]; the Scale Compiler Group also reports the implementation of a type-based method for alias analysis [25]. Probably closest to our approach, except for the level of generality, comes the idea of *semantic expansion* in the Ninja (Numerically Intensive Java) project where selected types are treated as language primitives; for complex numbers and arrays the generated Java code outperforms both C++ and Fortran [19,37]. Finally, we want to point out that rewriting techniques have long been used in code generator generators, e.g., Twig [1], burg [8], and iburg [7].

8 Conclusions and Future Work

In conclusion, the new, extensible simplifier optimizes user-defined classes and has been successfully integrated into the Gnu C++ front end. It coexists with the original simplifier and lifts several old, concrete rules to an abstract level, but also contains rules without counterparts in the original simplifier. The examples with the LiDIA and MTL libraries show the substantial performance gains that are possible even at the level of very-high intermediate representations, and hopefully contribute toward overcoming the performance concerns of those who choose non-object-oriented C or Fortran programming over object-based programming in C++ or write code that compromises readability for efficiency. As pointed out in the introduction we consider the simplification project to be the start of a new approach to optimizer generators, which handle built-in types and classes equally. For the next steps in that direction we want to get more experience with other high-level optimizations, investigating in particular the required analytical parts. A promising immediate next step seems to be to extend the type-based (alias) analysis of the current Gnu compiler [18] to user-defined types.

Acknowledgments. We thank Sun Chan, Alex Stepanov, and Bolek Szymanski for their encouragement and help in defining a suitable focus for the project; Sun was also directly involved in the first stages of the simplifier project. We furthermore acknowledge Fred Chow for insights and contributions in the early discussions on higher type optimizations. This work was supported in part by SGI, Mountain View, CA.

References

1. A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *JACM*, 23(3):488–501, 1976.
2. M. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1999.
3. S. Chiba. A metaobject protocol for C++. In *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, Oct. 1995.
4. CodeSourcery, LLC. *G++ Internal Representation*, August 2000. `http://gcc.gnu.org/onlinedocs`.
5. K. Czarnecki and U. W. Eisenecker. *Generative Programming—Towards a New Paradigm of Software Engineering.* Addison Wesley Longman, 2000.
6. K. Davis and D. Quinlan. ROSE II: An optimizing code transformer for C++ object-oriented array class libraries. In *Workshop on Parallel Object-Oriented Scientific Computing (POOSC'98), at 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1543 of *LNCS*. Springer Verlag, 1998.
7. C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator. *ACM TOPLAS*, 1(3):213–226, 1992.
8. C. W. Fraser, R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 4(27):68–76, 1992.

9. D. Q. Frederico Bassetti, Kei Davis. C++ expression templates performance issues in scientific computing. In *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, 1998.
10. D. P. Gregor, S. Schupp, and D. Musser. User-extensible simplification. A case study using MTL and LiDIA. Technical Report TR-00-7, Rensselaer Polytechnic Institute, 2000.
11. S. Z. Guyer and C. Li. An annotation language for optimizing software libraries. In T. Ball, editor, *2nd Conference on Domain-Specific Languages*. Usenix, 1999.
12. S. Haney, J. Crotinger, S. Karmesin, and S. Smith. PETE, the portable expression template engine. Technical Report LA-UR-99-777, Los Alamos National Laboratory, 1995.
13. D. Kapur and D. R. Musser. Tecton: A language for specifying generic system components. Technical Report 92-20, Rensselaer Polytechnic Institute Computer Science Department, July 1992.
14. D. Kapur, D. R. Musser, and X. Nie. An overview of the Tecton proof system. *Theoretical Computer Science*, 133:307–339, October 24 1994.
15. L.-Q. Lee, J. Siek, and A. Lumsdaine. The generic graph component library. In *Proc. of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99)*, volume 34, pages 399–414, 1999.
16. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.
17. S. B. Lippman. *C++ Gems*. Cambridge University Press, December 1996.
18. M. Mitchell. Type-based alias analysis. Dr.Dobb's Journal, October 2000.
19. J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM TOPLAS*, 22(3):265–295, March 2000.
20. D. Musser and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
21. D. Musser, S. Schupp, and R. Loos. Requirements-oriented programming. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming—International Seminar, Dagstuhl Castle,Germany 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2000.
22. D. Musser and A. Stepanov. *The ADA Generic Library: Linear List Processing Packages*. Springer-Verlag, 1989.
23. N. Myers. A new and useful template technique. In *C++ Gems* [17].
24. J. V. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA: A framework for scientific simulations on parallel architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.
25. Scale Compiler Group, Dept. of Comp. Science, Univ. Massachusetts. A scalable compiler for analytical experiments. `http://www-ali.cs.umass.edu/Scale`, 2000.
26. S. Schupp, D. P. Gregor, and D. Musser. Algebraic concepts represented in C++. Technical Report TR-00-8, Rensselaer Polytechnic Institute, 2000.
27. J. G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, Notre Dame, 1999.
28. 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.

29. C. Simonyi. The future is intentional. *IEEE Computer*, 1999.
30. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HP-94-93, Hewlett-Packard, 1995.
31. B. Stroustrup. *The C++ programming language, Third Edition.* Addison-Wesley, 3 edition, 1997.
32. The LiDIA Group. Lidia—a C++ library for computational number theory. `http://www.informatik.tu-darmstadt.de/TI/LiDIA/`.
33. T. Veldhuizen. Blitz++. `http://oonumerics.org/blitz`.
34. T. Veldhuizen. Expression templates. In *C++ Gems* [17].
35. R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, Dordrecht-Boston, 1982.
36. R. Wille. Concept lattices and conceptual knowledge systems. *Computers and Mathematics with Applications*, 23:493–522, 1992.
37. P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *Proceedings of the ACM Java Grande Conference*, 1999.