

Semantic and Behavioral Library Transformations

Sibylle Schupp,^a Douglas Gregor,^a David Musser,^a and
Shin-Ming Liu^b

^a*Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York 12180
{schupp,gregod,musser}@cs.rpi.edu*

^b*Hewlett-Packard, California
shin@cup.hp.com*

Abstract

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

1 Introduction

In the process of software development, software libraries have now become widely used. Many programmers have come to realize that the use of libraries results in code that is more robust and reliable than what they could have developed on their own from scratch. Even library designers themselves apply the principles of reusability to their software production and begin to stack libraries, e.g., to develop a graph library on top of a simple container library or a linear algebra library on top of an array library. The idea in all cases is to combine two or more independently developed library components and to plug them together in some particular way. In practice, however, the compositions are not always as seamless as one would wish.

Some problems in library composition are directly due to the design of component interfaces that simply do not fit. Interestingly, however, more subtle problems occur when a library has been explicitly designed with plug-in capabilities. If a library is

thoroughly parameterized, including even types and algorithms at implementation level, it can, on the one hand, easily be connected to other libraries just through (partial) bindings of its parameters. The loose coupling of components, on the other hand, cannot guarantee that all components are based on the same major design decisions or are designed with the same design principles in mind. It is thus possible to plug together libraries with conflicting design principles, and this is exactly the situation where features of a library may get lost or incur avoidable overhead in time or space.

Some resort to so-called lightweight languages, which provide quick ways to glue together software pieces that do not otherwise fit. But the glue mechanisms are based on little syntactic and no semantic analysis and therefore are restricted in their applicability; where they apply, they replace the original source code in an essentially unchecked and often ad-hoc manner. In contrast, the techniques of program transformation allow for systematic, semantics-based, and safe source code manipulations, which, moreover, have long proved useful in software engineering tasks related to the one at hand. Desugaring, optimization, and refactoring all are examples of well-established program transformations that can help overcome interoperability problems between software libraries.

At the same time, libraries differ from general software in two aspects that are interesting from the view of program transformation. On the one hand, as we will show, the interoperability problems of libraries almost always involve types and functions that are not known at the time the transformation system is designed. Systems for library transformation therefore have to be built in a way so that they lay out for example the sequence of steps for the required semantic checks independently from the constituents of a particular transformation. On the other hand, the users of these systems are library designers who differ from ordinary users with respect to their domain expertise as well as their readiness to make an initial extra effort if it results in an improved service for the clients of their library. A system for library transformations therefore can be designed so that library designers provide information that otherwise cannot be derived, and thus can operate under strong semantic assumptions.

Both extensibility and support from an expert user such as the library designer obviously are important characteristics of software not just when it comes to program transformation. In fact, both features have in many cases already been part of the design process of a library itself and as such have been formalized in the theory of library design. There, the term *concept* has proved to be a good abstraction to theoretically unify the types already implemented and any further extension of the library, by specifying their behavior in an algebraic way. It seems therefore natural to base the transformation of software libraries on a concept-theoretical foundation.

In this paper we introduce *Simplicissimus*, a concept-based framework for the transformation of (parameterized) libraries. *Simplicissimus* solves library-interoperabil-

ity problems at the expression level by formulating the desired change as a rewrite rule where the left and right hand side of a rule represent the expression and its replacement, respectively. *Simplicissimus* provides a framework for the specification of these rewrite rules, their validated application, and their execution. To summarize its features, it is an extensible, highly parameterized, and semi-automated module that depends on initial user information about the correctness conditions of a rewrite rule and the operands and operators it is applied to, but can then automatically select, validate, and apply the appropriate (sequence of) rules. The semantic checks are based on conceptual descriptions of formal and actual constituents of a rewrite rule on the one hand, and internally available concept hierarchies on the other hand. The current implementation of *Simplicissimus* interfaces to the GNU C++ compiler.

We begin our presentation with a characterization of the problems that occur in the composition of (parameterized) libraries, give evidence why these problems cannot be solved at the level of the libraries themselves or through redesigning them, and conclude from these examples some design principles for a system for library transformations (sect. 2). Moving from the requirements of a transformation system to their theoretical underpinning we summarize in section 3 the theory of concepts, discuss the role of concepts in the context of program transformation, and explain and illustrate concept-based transformation rules. Turning then to *Simplicissimus*, we first present its features in a language-independent way in sect. 4, then give an overview of the overall architecture, including the interface to the GNU compiler in sect. 5. We conclude the presentation with an extended example in sect. 6 and a discussion of related work in sect. 7. For a detailed explanation of the implementation in C++ and the user interface to *Simplicissimus* we refer to an earlier paper [1].

2 Transformation Kinds

Often, the need for program transformation comes from the need for system reengineering or even directly from flaws in the original design or implementation of a software system. In this section, however, we want to make the case that the composition of two libraries can cause problems even if each library by itself is well-designed. In illustration, we show three classes of common composition problems and use these problems to motivate some design requirements for a library transformation system.

2.1 Examples

A common source of problems comes from the two conflicting design principles of efficiency and safety. Most library designers give efficiency highest priority. They fear that their library will not be used if it cannot keep up with the most efficient, probably non-parameterized, implementation elsewhere and, with good reason, are concerned that all relative performance losses weigh even heavier if their library constitutes the foundation for applications on top of it. If given a choice in the libraries they use themselves, they therefore choose a fast implementation over a safe one. A standard example of the resulting problems their users then face are safe and unsafe vector (array) access functions. While for example the Standard Template Library (STL) provides both functions, a fast subscript operator on the one hand and the `at` member function, which does bounds checking, on the other hand, most high-performance libraries on top of it use the former function only and implement their access functions in terms of the fast but unsafe subscript operator. A user who would have preferred safety checks over speed is simply out of luck. Using a library transformation system, however, users can overrule the priorities of the library designer and temporarily replace, for example, the unsafe vector access function by the one that performs a boundary check.

Based on the same priority for efficiency, a second class of problems comes from the conflict between optimizations at the source code level and the resulting loss of readability. Especially in the area of numeric libraries where Fortran code still reigns, library designers have developed techniques that, anticipating optimization opportunities for compilers, result in code that outperforms Fortran code [2,3] but completely breaks with mathematical notational conventions. After the first euphoria about the performance results of these libraries users now are divided into two camps: those who value efficiency above all and those willing to sacrifice some of the efficiency gains in lieu of better readability and lesser maintenance costs. Again, library designers cannot serve both needs: they either apply or do not apply their optimization techniques. And again, program transformations can be used to conciliate two contradictory demands: when the optimizations do not take place directly in the source code, but are represented as rewrite rules and applied separately, it is possible to follow user conventions and to provide readable code without any performance loss.

Genericity itself, lastly, which is inherent in parameterized libraries, can cause problems due to the fact that the carrier library and its instantiation have been designed without any knowledge of each other. An instantiating library might follow different interface conventions than the carrier library, e.g., different naming conventions or different default bindings for arguments. Moreover, it might provide specialized functions that do not get used because the carrier library fails to invoke them. In practice, therefore, it frequently happens that optimizations get lost and that the composition of two high-performance libraries results in a non-competitive

library. The problem here is that it is obviously impossible for the carrier library to be prepared for special functions the existence of which is not yet known. With a subsequent transformation step, however, the library to be plugged in no longer depends on the preparations by the carrier library but can set up rewrite instructions to redirect an invocation to one of its own special functions.

2.2 *Design Requirements*

Whether a library lacks safe access to container elements, violates in its numerical code mathematical conventions, or fails to take advantage of optimizations possible under special circumstances, the problem in all these cases can be seen as a call to the “wrong” function and solved by replacing this functional expression by another one. Given at the same time that the transformations take place within the same language so that no language-specific differences in the bindings of variables, types, or scopes have to be addressed, it seems reasonable to restrict a system for library transformations to the transformation of expressions rather than the transformation of whole programs. Focusing on expressions restricts the possible source code manipulations to those that can be performed through rewriting function invocations, including operator, function, and procedure expressions as well as global functions and member functions. On the other hand, it reduces the overall complexity of the transformation system.

Secondly, the examples from the previous subsection show the need for generic transformation rules. If an application is safety-critical, for example, then certainly each unsafe container access function should be replaced by its bounds-checking counterpart. Likewise, if an operator-centric style for mathematical code is preferred, this style should include as many mathematical functions as possible. Determining exactly which functions do not comply with the desired policy, however, requires a pass through the entire library, including its implementation, which is a task as cumbersome as it is error-prone. To avoid the human user having to enumerate by hand each single transformation implied by a particular task, a transformation system should allow its users to specify in a generic way which transformation should be performed and which constraints hold for it and then automatically determine all specializations included.

Thirdly, looking more closely at the transformations discussed above, it is important to realize how little the three examples have in common: the first one deals with a kind of renovation for the particular case of bounds checking, the second one asks for syntactic sugar in the form of some two dozen operators, while the last one deals with classes of optimizations that are available only for certain combinations of types and algorithms. For a program transformation system it is thus impossible to anticipate which rewrite rules are needed or which libraries, types, or constraints might be involved. Instead, it should be organized as a framework so that it can

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

Lastly, it is worthwhile to note that the source and target of a transformation are not necessarily semantically equivalent. For example, replacing an unsafe vector access function by a bounds-checking one changes the behavior of the program, strictly speaking. Since apparently there is demand for this kind of transformation in practice, the library transformation system should not insist on semantic equivalence but should still ensure that any possible semantic change is well-controlled. Even a transformation as innocent as replacing a function by a clone that just contains additional debugging information requires guarantees that, in spite of overloading, partial specialization, and other ambiguity-prone language features, the intended function is the only one that gets replaced. The library transformation system should have mechanisms in place that prevent it from performing a transformation if these guarantees cannot be given.

To summarize, the examples and discussion above suggest the following principles for the design of a system for library transformations:

- expression-based transformations,
- generic transformation rules,
- user-defined rewrite rules and constraint checks,
- mostly semantics-preserving transformations.

2.3 *Interactive Transformations*

Whenever users participate in the development of a system they intend to use, it is important to organize the system so that information can be provided from a localized view: users specifying their own rewrite rules, defining constraints, or describing types should not need to know which other rules, constraints, or types have been defined by the system or other users. Likewise, if subsequent users extend the system in yet another way, they should not have to go back and revise their original specification.

Conversely, it is important to restrict a user's responsibility. Although there is no alternative to a human description of the semantics of a data type or a manual assertion of the appropriateness of a rule, passing the responsibility for correct transformations entirely back to the users is not only unsafe but also difficult to implement efficiently: the more dependence on the user, the less the translating compiler can enforce. To maximize the amount of checking that can be done automatically and at compile time, we therefore propose to follow a concept-based approach.

3 Concept-based Transformation Language

3.1 What are concepts?

The starting point for our discussion of concepts is the recognition that certain abstractions, such as (abstract data) types, have many distinguishing properties, such as existence of operations on them like \odot that themselves have certain properties such as commutativity ($x \odot y = y \odot x$) in the case of some abstractions (e.g., integer, real, or polynomial addition or multiplication) but not others (e.g., string concatenation or matrix multiplication). Furthermore, consideration of these properties is a useful basis for classification of a given universe of abstractions into subsets to determine, for example, for which types a certain algorithm will work correctly and efficiently. Although “concept” is sometimes used interchangeably with “abstraction,” we take the term to mean a *set of abstractions*, membership in which is determined by a given set of properties, called requirements. Formally, a concept consists of both a requirements set and an abstraction set:

Definition: A *concept* is a set R of *requirements* together with a set A of *abstractions*, such that an abstraction is included in A if and only if it satisfies all of the requirements in R .

The individual abstractions in a concept are sometimes called *models* of the concept. Besides concepts whose models are types, other important cases in computer science include mathematical function concepts and algorithm concepts, it being useful to distinguish between pure mathematical functions and recipes for their efficient computation. Concept refinement means adding more requirements, which means fewer abstractions (models) in the subconcept, but each is more specialized. This refinement relation yields *concept lattices*, whose general mathematical properties have been studied extensively by Wille [4,5] and later authors, but we do not need much of this theory for present purposes and to emphasize this we speak instead of working with *concept hierarchies*. Perhaps the most widely known example among software developers is the use of concept hierarchies in the C++ Standard Template Library (STL) documentation developed at SGI [6,7]. There, concepts are used to classify container, iterator, and function object type abstractions so that one can document precisely which types one can pass to the generic algorithms in the library and be guaranteed of correct and efficient execution. A diagram of a portion of the STL container hierarchy is shown in Figure 1. In a similar way, algorithm concepts appear to be an excellent means of documenting algorithms for end-users. A well-developed algorithm concept hierarchy may also be a good way of organizing compile-time or run-time databases of performance data and metadata for use in various code optimization strategies [8].

Regarding development of mathematical function concepts, one can of course point

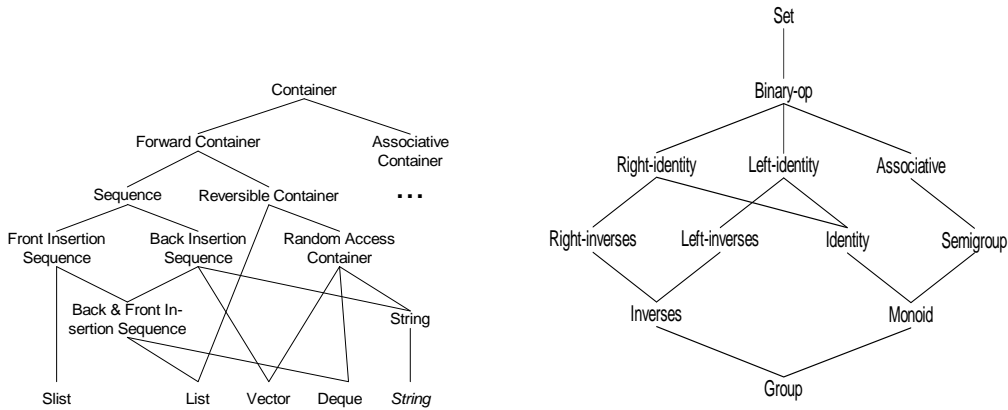


Fig. 1. Diagrams of two concept hierarchies: On the left, a portion of the STL container hierarchy (based on the SGI STL documentation [6,7]). On the right, a hierarchy of standard algebraic concepts.

to the whole history of abstract algebra for examples. Concepts developed by refinement include SEMIGROUP, MONOID, GROUP, RING, and FIELD, although the term “algebraic structure” is more frequently used. The diagram on the right in Figure 1 depicts a hierarchy of such algebraic concepts. Such concepts have long played an important role in the design and justification of many mathematical software packages, from Fortran libraries to general mathematical systems like Mathematica, Maple, or Axiom. In a few cases algebraic concepts have been employed more directly in actual implementations of mathematical software, most notably in Axiom, which allows algorithms to be parameterized by such concepts, e.g., operations on polynomials over any commutative ring with identity.

In our *Simplicissimus* work we are using algebraic concepts to help classify transformation rules and validate their application to expressions and control constructs involving specific types. The use of concepts may be seen to take place at three levels—theoretical, implementation design, and the actual implementation of *Simplicissimus* in C++.

At the theoretical level, we have expressed many algebraic and other concepts that underly program transformations in the Tecton concept description language [9,10]. Tecton is a small language specifically for the purpose of specifying and reasoning about generic software components. Each Tecton concept description typically contains a `requires` clause that lists axioms that must be satisfied for an abstraction to belong to the concept. These axioms are in addition to those that are imported from other concepts, making the concept described a refinement of those other concepts.

At the implementation design level, we mirror much of the hierarchical organization of the Tecton concept descriptions in the way we organize and use program transformations. First, we have no long lists of transformations but instead have

a hierarchy of class definitions containing many small groups of transformations that are accessed and applied (or rejected) based on type and concept checking and semantic analysis. Second, the axioms of the Tecton descriptions translate into *conditional rewrite rules*, which are the main form of transformation implemented.

Finally, at the actual implementation level, these design level decisions are realized in terms of C++ features. We retain the essential features of the design level, including most importantly the organization of transformation rules by means of template classes corresponding to concepts. Other decisions at this level are less essential. For example, we use *expression templates* to represent rewrite rules and target expressions at compile time. This and other highlights of the current implementation are discussed in section 5. Here we expand mainly on the implementation design level.

3.2 Concept-based transformation rules

Suppose we are given a simple axiom, such as $x + 0 = x$, which we would like to apply as a rewrite rule $x + 0 \rightarrow x$, but not only for a specific type like `int` but for *any* type on which `+` and `0` are defined with semantics that include the given equation. We thus associate the equation and the rewrite rule with the concept with the minimal set of requirements that enable its correct application. Therefore it can be correctly applied to expressions over every type in the concept, including not just those that might be known at the time the rule is designed but also any user-defined types that can be shown to belong to the concept (by showing all of the concept's requirements are met). For example, the given right-identity simplification rule, which is repeated in fig. 2, comes directly from one of the equations of the MONOID concept. Although stated in terms of `+` and its identity element `0`, these are actually parameters, so that the rule also represents, for example,

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

The second rule in fig. 2 is another algebraic example from the RIGHT INVERSES concept and the third is a rule for transforming uses of an unchecked subscripting operator `[]` into calls of a member function `at` that does bounds checking. In the C++ standard library the `vector`, `deque`, and `string` containers all provide these options for subscripting, but rather than having a separate rule for each of these containers or any newly invented container we write a single rule for a BOUNDED ACCESS CONTAINER concept characterizing all containers that provide both functions with the stated relationship between them. This example is discussed further in sect. 6, where we describe the kinds of rules that library developers might profitably devise for use by Simplicissimus for transformations of their libraries. Devising rules such as those shown here for the MONOID and RIGHT

$x + 0$	$\rightarrow x$	$x : T, (T, +, 0) \in \text{MONOID}$
$x + (-x)$	$\rightarrow 0$	$x : T, (T, +, -, 0) \in \text{RIGHT INVERSES}$
$c[i]$	$\rightarrow c.\text{at}[i]$	$c : C, i : I, (C, I, [], \text{at})$ $\in \text{BOUNDED ACCESS CONTAINER}$

Fig. 2. Concept-based rules (left) and the requirements for their parameters (right)

INVERSES concepts would not be the province of library developers; rather, they are included in the base of algebraic concepts by which *Simplicissimus* organizes simplification rules for optimization of built-in and user-defined types.

Concept-based transformation rules require additional capabilities that a more traditional (type-based) program transformation system does not support, such as the ability to determine if a given type models a concept and to apply a concept-based rule given a specific set of types. To explore the design and implementation of a concept-based program transformation system we developed the *Simplicissimus* program transformation framework.

4 *Simplicissimus*

Simplicissimus is an extensible program transformation system based on the application of conditional rewrite rules to expressions. Users extend *Simplicissimus* by introducing new conditional rewrite rules, types, operations, and additional semantic information used in rewrite rule conditions. Additionally, users may modify or extend the behavior of *Simplicissimus* at any point where the framework must make an arbitrary decision, e.g., to choose between several rewrite rules (that all apply) or decide when to stop applying rewrite rules.

The input to *Simplicissimus* can be divided into four parts: expressions to be transformed, semantic information about types and operations, conditional rewrite rules, and behavioral customization code. Expressions to be transformed by *Simplicissimus* are mapped from their internal representation in the compiler to a compiler-independent representation used internally for *Simplicissimus*; this mapping, although aided by user input, is an implementation detail that will not be discussed here. The input of type and operation semantic information is described in sect. 4.1, the form and application of conditional rewrite rules are described in sect. 4.2, and the customization of program transformation strategies in sect. 4.4.

4.1 User-defined Semantic Information

User-defined conditional rewrite rules often require additional semantic information. Some semantic information, such as freedom from side effects in an operation, is so common that it is considered a basic property of the operation; for these ubiquitous traits, the user need only fill out a simple pattern.

Most semantic information, however, is naturally domain- or application-specific. Such information may not fit well with any predefined properties or concept models, and therefore users are allowed to introduce semantic information by attaching such information to types and operations in a declarative manner. For instance, the user may state that indexing a C++ `vector` is a *bound-unsafe* operation, meaning that it does not check that the index is within bounds; such a declaration could be made by introducing a *bound-safety* property, which for the `vector` indexing operation would have the value *unsafe* but for the corresponding safe routine `at` would have the value *safe*. Then the conditions on a rewrite rule can invoke a query on this property to decide whether or not a specific rewrite rule is applicable.

4.2 Matching Rewrite Rules

To determine if a given conditional rewrite rule may be applied to an expression, we must try to match the syntax of the rewrite rule to the syntax of the expression and then determine if the semantics of the expression meets the requirements of the rewrite rule. The syntactic matching is merely first-order unification: we let *nullary*, *unary*, *binary*, etc., be function symbols and let operations and expressions be terms. Thus, the product of `x` and `1` is represented by the term `binary(*, x, 1)`. We then let words beginning with uppercase letters denote variables (in the unification sense), so that we can represent the left-hand side of a rule as a term with zero or more variables in it. Thus, the left-hand side of the right-identity simplification given in fig. 2 is written `binary(Op, X, Zero)`. The syntactic matching of this right-identity rule to the product of `x` and `1` is the solution to the one-sided unification problem (“matching”)

$$\text{binary}(*, x, 1) \stackrel{?}{=} \text{binary}(\text{Op}, X, \text{Zero}),$$

which will succeed with the variable bindings $\{\text{Op} \mapsto *, X \mapsto x, \text{Zero} \mapsto 1\}$.

The second stage in the matching of an expression to a rewrite rule is to validate the condition based on the variable bindings. In the case of the above right-identity simplification rule, we require that `*` be a MONOID operation with identity `1` and that application of `*` has no side effects. A predicate function coupled with the rewrite rule checks these conditions, using a simple functional language (written as a subset of the source language) with access to any semantic information provided

by the framework or by the user.

4.3 *Concept inference*

Semantic checks in concept-based transformation rules require matching the concrete types used in an expression to the concept requirements of the concept-based rule, a process called *concept inference*. Concept inference must account for refinement relationships between concepts because actual data types directly model very specific concepts (e.g., an integer type models an INTEGRAL DOMAIN) whereas concept-based rules might be associated with more general concepts (e.g., the right-identity rule in Fig. 2 is associated with a MONOID). To apply a right-identity rule to an expression $i + 0$, where i is an integer type, the semantic check must determine that the INTEGRAL DOMAIN concept refines the MONOID concept; given this refinement relation, it is determined that the integer type does in fact model a MONOID, and the right-identity transformation rule can safely be applied.

The mechanics of concept inference are directly related to the structure of concept hierarchies, such as those in Fig. 1. A type will explicitly identify the most specific concepts that it models, e.g., an INTEGRAL DOMAIN or a VECTOR. Concept inference is initiated at each concept modeled by the type in question, and moves upwards in the concept hierarchy (from specific concepts to more general ones) searching for a match.

A match between a concept that is required for a transformation rule and one found in the concept hierarchy depends both on the equivalence of the concept name and on the values, types, and operations used in the concept. The MONOID concept, for instance, uses a type T , a binary operation $+$, and a literal value 0 (of type T). All information about values, types, and operations is available within the concept hierarchy, but the information extracted from a particular instance of a concept-based rule may be incomplete, because the information is extracted from the left-hand side of the rewrite rule during (one-sided) unification. Given an expression $x + (-x)$ that matches the right-inverses rewrite rule shown in Fig. 2, we can deduce only that the rule requires the type of x to model RIGHT INVERSES with a binary operation $+$ and unary operation $-$. Simplicissimus matches the complete concept instance in the concept hierarchy with the (possibly incomplete) information derived from the concept-based rule through one-sided unification: the known information must agree, and the unknown information is returned from the concept inference step via the list of variable bindings.

The effectiveness of a program transformation system based on rewrite rules depends directly on *what* rules are applied, *when* rules are applied to expressions and their subexpressions, and *how* we choose between applicable rewrite rules. Fixing a strategy to answer any of these questions inevitably biases the framework in favor of a certain set of transformation problems to the exclusion of others. *Simplicissimus* therefore seeks to enable users to devise their own strategies and organize them in a manner that supports the application of a different strategies at different phases. The remainder of this section discusses the customizability of *Simplicissimus*' rewrite system.

Ambiguity resolution The choice of a rewrite rule at any particular step may be ambiguous, with the expression satisfying the syntactic and semantic requirements of two or more rewrite rules. Confluence of the rewrite system cannot be guaranteed in the general case, so the choice between rewrite rules may affect the optimality of the final solution. Such decisions cannot be made in an arbitrary manner by the implementation, but instead are left to user-defined “arbiters” that determine what rewrite rule is most appropriate in a given situation.

Arbiters are free to use any additional information that may be attached to certain expression or rewrite rules, e.g., an estimate of the run-time cost of an operation or an annotation that a particular rewrite rule transforms an unsafe, unchecked operation into a safer (albeit more expensive) checked operation. One such metric—the run-time cost of computing the value of an expression—is a basic trait of every operation in *Simplicissimus*. An arbiter can determine if the application of a particular rewrite rule to an expression will increase or decrease the cost of computing that result.

Arbiters themselves are functions mapping a set of rewrite candidates and a source expression into a target expression. The target expression may be the result of rewriting the source expression via one of the rewrite rules or, alternatively, will be the source expression if no candidates satisfy the arbiter's criteria.

Simplicissimus defines several simple arbiters that are adequate for the needs of most users. The *first-fit* arbiter blindly applies the first valid transformation, ignoring any efficiency information. The *first-best-fit* arbiter applies the transformation resulting in the largest decrease in the cost of computation, based on the cost estimation provided with every operation; in the case of a tie, the first rewrite rule with minimal resulting cost is selected. A related arbiter, *last-best-fit* differs only in that it will choose the last rewrite rule with minimal resulting cost in the case of a tie. Neither *first-best-fit* nor *last-best-fit* will select a transformation that will increase the cost of computing a particular result.

Traversal order When it is not feasible to apply rewrite rules to an expression until

an irreducible expression is attained, e.g., for efficiency reasons, the traversal order becomes a factor in the effectiveness of the transformation system. A bottom-up transformation may perform well for a certain set of transformations, whereas a top-down approach may be more appropriate for an entirely different transformation set. Traversal order in *Simplicissimus* is therefore user-defined, and is controlled by “directors.”

Directors, like arbiters, are free to use any information that can be deduced from the expression, including built-in properties (such as the “has side effects” predicate) and user-defined, library-specific properties. Directors can initiate transformation of an expression guided by any director/arbiter combination and on any set of rewrite rules. Directors collect the results of these transformations and may themselves rewrite expressions, e.g., by substituting the transformed subexpressions of an expression to determine the final transformed expression.

Simplicissimus supplies two simple directors that cover simple traversal strategies. The *bottom-up* director performs a bottom-up traversal by first (recursively) transforming the children of each expression, substituting the transformed children into the expression, and then transforming this expression. The *top-down* director performs a top-down traversal, where top-level expressions are transformed first and then subexpressions are transformed and substituted into the top-level expression.

With the flexibility of defining a traversal order comes the responsibility of guaranteeing termination. Both directors supplied with *Simplicissimus* guarantee termination because their transformations are limited to one rule application per expression. However, a more aggressive director—that, for instance, applies rewrite rules until an irreducible term is found—must rely on the behavior of the arbiter(s) involved or on the set of rewrite rules. Practically speaking, the use of arbiters that enforce a strictly decreasing expression cost will guarantee termination.

Stages The best ambiguity resolution strategy for a given set of rewrite rules may not coincide with the best ambiguity resolution strategy for a different set of rules. For instance, we may choose to supply safe container access but optimize arithmetic operations for efficiency. While it is indeed possible to achieve this goal using a single arbiter, the result is wasteful and overly complex. Instead, *Simplicissimus* provides the notion of a *stage* in the program transformation process.

A stage is a set of rewrite rules coupled with an arbiter and a director. The director manages the traversal of the expression within the stage, and the arbiter resolves ambiguities among rewrite rules in the stage. Transformation of an expression in the *Simplicissimus* framework is performed by passing an expression through a stage transformation pipeline. Stages are arranged linearly by the user, and the result of applying the transformations in a given stage is passed as the input to the next stage. The output of the final stage is the final transformed result.

The construction of individual stages and of the stage pipeline requires specific

knowledge of the interaction of rewrite rules and arbiters, even across stage boundaries. It is therefore unrealistic to expect the average user to construct each of the stages and the pipeline prior to using *Simplicissimus*. We have therefore identified a three-stage pipeline that is applicable in many program transformation instances. Most rewrite rules can be classified in one of the three stages: *normalization*, *simplification*, or *specialization*.

The normalization stage includes transformations that rewrite expressions into a canonical form. For instance, there may be ten semantically equivalent ways to access a particular element within a data structure, one of which is chosen to be the canonical form. Rewrite rules in the normalization stage will transform these other nine forms into the canonical form, so that later stages need only consider the canonical form. The normalization stage is based on the *first-best* arbiter, and therefore does not require that its rewrite rules decrease the computation cost of the expression.

The simplification stage includes transformations that rewrite (canonical) expressions into simpler (also canonical) expressions. The *first-best* arbiter is employed to compare the computational benefits of each transformation.

The specialization stage can be seen as the inverse to the normalization stage. The rewrite rules in this stage transform an expression in canonical form into a specialized form that may be more suitable (e.g., more efficient or more safe) for the given task.

Search space pruning A typical program transformation system may comprise a hundred or more abstract rewrite rules, and the cost of checking the applicability of each of these rules at each subexpression can be quite high. We therefore seek a method to prune the number of rewrite rules we must examine at each subexpression to reduce the overall cost. Automatic search space pruning, however, requires knowledge of the syntactic matching and/or semantic checking of all rewrite rules within a system; information that is not available in the *Simplicissimus* framework.

Instead, we provide an explicit rule grouping construct that allows the author of rewrite rules to group related rules under a set of assumptions that the grouping construct can guarantee. For instance, a software library may have a large number of rewrite rules that are specific to the data types in that library. The author of these rewrite rules can create a rewrite rule group that will only be considered when one of the operands in the expression to be transformed uses a type from that library. Therefore, *Simplicissimus* can ignore all rewrite rules specific to that library when that library is not in use.

Rule grouping constraints are similar to rewrite rule constraints. They may have both syntactic and semantic checks, and it is assured that a rule that is only accessible within a group will only be checked against an expression if the expression has passed the syntactic and semantic checks of the group. This cascading check-

ing policy, along with the ability to nest rule groups, enables the transformation of the rewrite rule search space from a linear search space into a tree. Each node in the tree introduces additional constraints that are passed on to its children, thus avoiding redundant checking of expression characteristics.

Non-conservative extensions The correctness of a *Simplicissimus* transformation corresponds directly to the correctness of each rewrite rule applied. Rewrite rules, in turn, rely on the correctness of user-supplied information about the semantic behavior of user-defined data types. A user is free to introduce “non-conservative” rewrite rules that transform an expression into a *nearly* equivalent expression that has more desirable behavior. Sect. 6 describes one such transformation that replaces function calls to an unchecked data structure access function into calls to a checked access function. The semantics of transformed expressions differ from the original expressions in that they may throw an exception if the data structure is incorrectly accessed. In areas where the efficiency of generated code is of primary importance, rewrite rules may make more aggressive assumptions, such as the non-aliasing of operands in a matrix-matrix multiply routine.

5 Implementation

The *Simplicissimus* framework consists of three parts: the core engine, an interface to the translating compiler, and user-provided information. In this section we summarize each part.

The core engine is a stand-alone C++ program of about 19k lines of code that takes an arbitrary expression and returns its transformation according to a set of (user-defined or built-in) rewrite rules. As explained in the previous section, this rewrite process includes pattern matching, concept inference, and the look-up of user-provided information. The core engine assumes that both the actual expression and the expressions on the left and right hand side of a rewrite rule are represented in *expression template* format [11]. An expression template, in short, is a C++ class template that represents an expression in prefix form, annotated with meta-information about its arity, and, most importantly for genericity, with parameterized operands. Fig. 3 shows the representation of a subscript expression as an expression template.

```
Expr< BinaryExpr<Subscript, Vec, Index> >
```

Fig. 3. (Binary) expression template, representing the subscript expression

Expression templates, introduced in C++ for numerical optimizations, have become an important idiom for other kinds of computation, especially ones based on partial evaluation. The main motivation for using them as the intermediate representa-

tion for *Simplicissimus*, however, is that two major parts of the core engine, pattern matching and recursive expression walking, are already implemented by the translating compiler to process template instantiation, and can therefore simply be reused.

Although expression templates are legal C++ code, source code at the user level typically is not written in expression template format. It is necessary, therefore, to transform normal C++ expressions into equivalent instances of expression templates. Given the complexity of the C++ language, however, handling source code to its full extent is not easy. To avoid the effort of parsing C++ code, a tedious and non-trivial task, we decided to use full-ædged compilers and provide interfaces to each compiler’s internal representation at the level of the abstract syntax tree. Currently, we have implemented an interface to the GNU compiler, *gcc*; proprietary restrictions aside, other front ends could be interfaced as well. The GNU interface consists of two files that provide mappings between two internal representations: from an abstract syntax tree in the internal TREE language of *gcc* [12] to *Simplicissimus*’ internal expression templates and, conversely, from expression templates back to TREE expressions. In a relatively straightforward way the interface module recursively walks the tree in one representation and writes out its counterpart in the complementary representation. For the conversion to expression templates the interface contains the functions listed in Fig. 4. There are similar functions for the simpler task of converting an expression template back to TREE format.

```
tree build_expr_template_from_variable( tree)
tree build_expr_template_from_literal( tree)
tree build_expr_template_from_call( tree, tree (*)(tree))
tree build_expr_template_from_unary_op(
    tree, tree (*)(tree))
tree build_expr_template_from_binary_op(
    tree, tree (*)(tree))
tree build_expr_template_from_convert_expr(
    tree, tree, tree (*)(tree))
```

Fig. 4. Interface functions between *Simplicissimus* and GCC

Fig. 5 depicts how *Simplicissimus* and the GNU compiler collaborate. The figure also shows that the process of expression transformations is entirely transparent to application users: provided their library designer has set up the required semantic information they can control *Simplicissimus* simply with a command line flag:

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

The semantic information itself is encapsulated in *traits*, which are interface templates that provide a “convenient way to associate related types, values, and functions with a template parameter type without requiring that they be defined as members of the type” [13]. Since traits can provide uniform interfaces on the one hand and encapsulate specific behavior on the other hand, they were originally used to

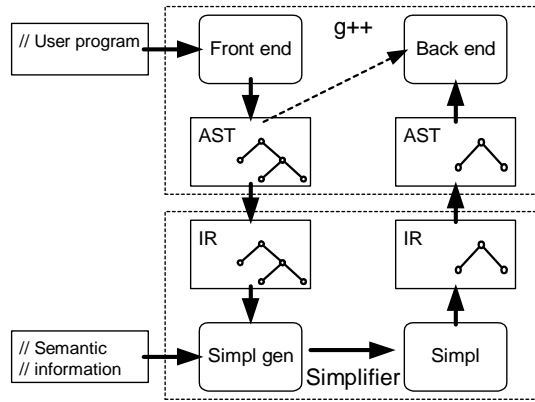


Fig. 5. A user expression, translated into GNU’s abstract syntax tree (upper box), mapped to the internal representation in Simplicissimus, rewritten by the core engine (lower box), and mapped back.

overcome problems with the internationalization of C++. Traits are now part of the language specification of I/O classes, strings, floating points, characters, and several other built-in and library types [14]. In Simplicissimus, they allow the core engine to express the semantic check independently from any particular type and to abstractly lay out the various steps of the semantic check. Besides, they help reduce run-time cost since the information stored there can be looked up at instantiation time.

```

class BinaryOpTraits<Subscript> {
public:
    typedef __true_type    is_applicative;
    typedef __false_type  has_side_effects;
    typedef __true_type    lhs_is_const;
    typedef __true_type    rhs_is_const;
    static const int cost = SUBSCRIPT_COST;
};

```

Fig. 6. Trait, describing the computational behavior of the subscript operator

The use of traits is extended to represent additional, more complex information about types and operations. For any given type, the concepts the type models are listed via a set of traits and the concepts themselves are traits. Concept refinement is modeled directly in C++ as (multiple) inheritance, and we reuse the compiler’s ability to transform a derived (sub-)class into one of its base (super-)classes to model concept inference (see sect. 4.3).

C++ templates can be viewed as a simplistic functional programming language, where all computation is performed via template instantiation at compile-time. Template specialization, which allows the user to replace the definition of a parameterized class with a more specific definition based on the values of its parameters, serves as a basic control structure. The whole of the Simplicissimus core is implemented based on template specialization, using a well-established technique called

template metaprogramming. For more information we refer the interested reader to the literature and programming examples elsewhere [15,16].

Template metaprogramming allows customizations to *Simplicissimus*, such as new ambiguity resolution strategies introduced via arbiters or new expression traversal strategies introduced via directors, to be implemented as compile-time programs that integrate seamlessly with the *Simplicissimus* core. In fact, the default behaviors described in sect. 4.4 are implemented via template metaprograms that enjoy no more privileges than any extension to *Simplicissimus*.

6 Extended Example

Now, exactly what do users have to do to utilize *Simplicissimus*? In this section we take up an example we discussed already earlier, safe vs. unsafe subscribing of containers. We already claimed that, assuming a library using the (unsafe) `[]` operator and an application on top of it requiring safe subscribing, a user can make the underlying library `£t` by introducing a rewrite rule that replaces unchecked subscribing (with operator `[]`) by bounds-checked subscribing (with a member function, `at`). This rule, moreover, if designed at the right conceptual level, suffices to replace all unsafe occurrences of subscribing in a systematic manner, without the need for any additional user intervention. Neither do the users need to know which types the underlying library implements nor do they have to explicitly enumerate all types the rewrite rule applies to. We now show how to define such a rule and integrate it into *Simplicissimus*. In short, the user's task breaks down into at most the following subtasks:

- defining the rewrite rule `[] → at`, including the conditions under which it is applicable;
- defining the concepts in which the rule is expressed;
- describing the behavior of the two operators `[]` and `at` involved in the rule;
- providing a compiler-specific wrapper for each operator.

Not all four steps, however, are always necessary. In many cases the last three steps need not be done since the required information has been provided earlier: if the operators `[]` or `at` are part of another rewrite rule already available, the operator description and the wrappers given there can be re-used; likewise, if the conditions for the rewrite rule depend on concepts already defined, there is no need to repeat their definition. In fact, the fully-developed *Simplicissimus* module will provide both concept definitions and operator descriptions for all types and classes of the C++ language and its standard libraries. For the sake of the example, however, we pretend that the operators `[]` and `at` are yet unknown. The following subsections illustrate first how to design a rule, then discuss the operator description required, and finally motivate the concept of a `BOUNDED ACCESS CONTAINER` for the rule

under consideration. The explanation of the wrapper function we omit; instead we refer to the *Simplicissimus* intermediate representation [17].

6.1 Defining a rule

Rules are implemented as classes that hold the left and right hand side of a rewrite rule along with its conditions, all three parts of which are expressed in *Simplicissimus*' internal expression-template language. In this internal language, as shown before (fig. 3), a `[]`-expression is represented as

```
Expr< BinaryExpr<Subscript, Cont, Index> > >
```

and an `at`-expression as

```
Expr< BinaryExpr<At, Cont, Index> > >.
```

As also explained before, there exists a mapping between a C++ expression and the corresponding expression template, which can be applied almost mechanically and, syntax aside, provides no difficulties.

Suppose now we implement the rule `[]` \rightarrow `at` as a class called `SubscriptToAt`. Defining this class requires the definition of two auxiliary classes, `Result` and `Validator`. Both are small classes that encapsulate exactly one member each. The class `Result` holds the right-hand side of the rule. Its body introduces a type definition, `result`, that aliases the right-hand side of the rule, in expression-template format. In our example, the value of `result` is the expression template above, `Expr< BinaryExpr<At, Cont, Index> > >`. The class `Validator` introduces a boolean variable `valid` and a conjunction of boolean expressions that specify the conditions of a rule, and sets the value of `valid` to the value of this boolean expression. In our example, as we show later, the correctness of the rule depends on the property of a bounded container. We therefore set `valid` to the value of the predicate `is_bounded_container`. Both auxiliary classes, `Result` and `Validator`, are parameterized by the left-hand side of the rewrite rule (in the expression-template format above), and thus are instantiated only when the rule syntactically matches. Conversely, if the instantiation fails, there is no need to check the constraints of the rule and no way to access its result.

Grouped together, `Result` and `Validator` essentially form the class `SubscriptToAt`, that is, provide all information *Simplicissimus* needs. For an arbitrary C++ expression, *Simplicissimus* is now able to determine whether it matches the rule defined and satisfies its constraints; if so, it can return the result. All that is left to do, therefore, is to add the new rule to the set of rules already known to *Simplicissimus*. The complete code of the class and the directive to include this class can be found in the appendix A.

6.2 Describing Operators

With the last lines of the previous section, the new rule has become available in the system. The `at` and `[]` operators themselves, however, are not automatically known to the system. Unless someone else has already done so, we have to provide a description of their behavior. For this purpose, `Simplicissimus` prepares operator trait classes for each arity. Fig. 6 lists an example of an instance of `BinaryOpTraits`; there also exist `UnaryOpTraits`, `TernaryOpTraits`, etc. Each operator and function is supposed to define a trait of its respective arity.

The varying number of arguments aside, all traits gather the same kind of information:

- the result type;
- information about the run-time cost of an operation (`cost`)
- information about the possible change of state (`operand_is_constant`, `is_applicative`, `has_side_effects`).

In addition, each trait includes a function `apply`, internally used during the conversion from expression templates back to “normal” expressions, that invokes the original operator.

Traits are important for the correct definition of the `Validator` classes discussed. Without them it would not be possible to write, e.g., validators that require protection against certain exceptions or state changes. Although the validator class in our example is relatively simple and `Simplicissimus` in fact only needs the member `apply`, it is at least good style to provide a complete definition of the traits classes for `[]` and `at`.

Technically speaking, writing a trait class comes down to filling out a template. Since the features of interest are fixed, a user only has to go through this set and to decide for each feature whether or not it applies to the given operator, i.e., to ask whether this operator is applicative or not, has side effects or not, etc. On the other hand, the correct answer to each question requires good knowledge of the specification of a function. Moreover, due to the nature of the information, each definition necessarily has to be accepted unchecked: whatever a user defines in a trait, `Simplicissimus` has to consider as correct.

In our example, one needs to define a `BinaryOpTrait` each for the operator `[]` and its counterpart `at`. The simplified trait for `[]`, with all information correctly filled in, is listed in Fig. 6; the full code of the trait can be found in the appendix B. Since `at` enjoys the same behavior as `[]` with respect to the information requested, the body of its trait is identical to that of `[]`. We therefore do not list it separately.

6.3 Defining Concepts

Although it is desirable to express a conditional rewrite rule in terms of standardized concepts or at least concepts already available in `Simplicissimus`, this is not always possible. In our example rewrite rule, for instance, it is necessary to introduce a new concept, the `BOUNDED ACCESS CONTAINER`.

Defining a new concept typically is done as a refinement of an existing one. In case of the `BOUNDED ACCESS CONTAINER`, it is possible to refine the `RANDOM ACCESS CONTAINER` concept (as defined in `SGI STL` [7]). The latter concept already requires the operator `[]` be defined for any type C belonging to the concept. It furthermore gives a semantic definition of the operator, in terms of other members, and additionally specifies the computing time to be amortized constant. There is, however, no requirement that the `[]` operator do bounds checking—the meaning of $c[i]$ is undefined if $i < 0$ or $i \geq c.size()$. Nor is there a requirement for a separate function that does bounds-checked accesses. We thus include a requirement in `BOUNDED ACCESS CONTAINER` that any type C belonging to this concept must provide a member function `at` such that

$$c.at(i) = c[i]$$

whenever $0 \leq i < c.size()$; otherwise, the function call $c.at(i)$ must raise an exception. Further, we require that `at` also be an amortized constant time operation. Next, consider the types that may or may not belong to `BOUNDED ACCESS CONTAINER`. Obviously, the vector, deque, and string types provided in the C++ standard library belong to this concept, according to the requirements imposed on them in the standard. On the other hand the STL map container does not belong; although it has an operator `[]`, it does not have a corresponding `at` member function, and could not sensibly have one added to it since there is no reasonable interpretation of “bounds” on its keys for nonintegral key types. (The map container is not even a model of `RANDOM ACCESS CONTAINER`, since `[]` is not an amortized constant time operation.) It is important to stress, however, the set of types belonging to `BOUNDED ACCESS CONTAINER` is not just restricted to ones listed above. If a user defines a new type (by means of a class definition) in which the operator `[]` and a member function `at` have the required semantics and complexity, this new type will model a `BOUNDED ACCESS CONTAINER` just as do the examples from the standard library.

Now consider the transformation rule from figure 2,

$$c[i] \rightarrow c.at[i]$$

where $c : C$, $i : I$, and $(C, I, [], at) \in \text{BOUNDED ACCESS CONTAINER}$. By the definition of `at`, this rule transforms accesses that are within bounds into equivalent accesses using `at`, and it transforms undefined out-of-bounds accesses into

ones that cause `at` to raise an exception. This concept-based transformation will be applied to any expression $c[i]$ in which c is of a type C identified as belonging to `BOUNDED ACCESS CONTAINER`, but it will not be misapplied when c is a map container or a user defined type in which not all of the semantic requirements of the concept are met.

7 Related Work

Much work is going on in applying ideas from program transformation to large software systems, both of legacy systems and of recent component-based ones. The TXL programming language and rapid prototyping system [18,19] performs transformations that are based on a BNF-description of the source language and a set of *by-example* rules. As a language and compiler of its own, and unlike `Simplicissimus`, TXL can handle cross-language transformations and has been applied to several major programs, for example Cobol, PL/I, and RPG code for the Year 2000 bug. Another transformation system of significantly greater power than `Simplicissimus` is the Domain Maintenance System (DMS) [20,21]. Built on the theory of *transformational design*, DMS overcomes the separation of construction and maintenance through a sequence of transformations that ultimately synthesize code from a specification. Of all transformation systems we aware of, DMS gives highest priority to scalability and aims at the transformation of systems of several hundred-thousand or million lines of code. While DMS integrates the whole software development cycle, the optimizer generator OPTIMIX operates on the intermediate representation of a program, its control-flow graph, and specifies and performs program optimizations as graph transformations [22,23]. Graph rewriting is on the one hand more difficult than the rewriting of expression trees that we perform and is capable of reasoning over data-flow properties [24], which is difficult to do with programs represented in expression tree format. To our knowledge, however, current graph rewrite systems are restricted to predefined types, and thus act at a lower level of abstraction than our approach. The Stratego system [25], finally, is a system based on rewriting strategies, that is, rewrite-based algorithms for term transformations. Stratego primitives are (generic) iteration and traversal strategies, which give users control over the application of rewrite rules and their order. In the Code Boost project [26] for numeric (coordinate-free) programming, Stratego is used to eliminate temporaries in C++, an optimization `Simplicissimus` also has been used for. The Code Boost framework, however, restricts its C++ handling to the subset relevant to the Code Boost optimizer, while one of the major design decision of `Simplicissimus` was to ensure the ability to process the full C++ language.

From a more theoretical point of view, the approach in `Simplicissimus` borrows from ideas in program specialization and partial evaluation [27], while from a software-engineering view it bears similarities to modern software methodologies, most notably intentional programming, subject-oriented programming, adaptive

programming, and aspect-oriented programming [28–32]. Probably closest to our approach is aspect-oriented programming (AOP), insofar as it advocates a software architecture where the “cross-cutting aspects” are cleanly modularized and a “weaver” combines aspects and components into executable code. In fact, our core engine can be viewed as a weaver. The major conceptual difference comes from our application to libraries or, more generally, from the situation where software designer and software user are temporally separated. Misusing the client-server terminology, one could say that we rewrite code on the client side, while AOP rewrites it on the server side. For non-parameterized data types this difference does not matter; for parameterized types, however, the particular rewrite typically depends on the particular instantiation of a type, and thus cannot be dealt with on the server side.

Extensibility, at the same time, is a trend in compiler construction. The Broadway compiler, for example, can be controlled by an annotation language for optimizing libraries [33]. Closer to a mainstream language but limited to one type, the ROSE C++ source-to-source preprocessor [34] is extensible by a small generator for optimization specifications of class arrays. Finally, the system Magik [35] performs code optimization but can also incorporate into the compiler the meaning of library interfaces and data structures, thus allowing them to be checked for errors to the same degree as for built-in constructs. A key difference of our approach lies in our organization of optimizing transformations in terms of concepts, which allows a library designer to augment a library with many transformations and enable them for any new type the library defines merely by identifying the most general concept to which the type belongs.

8 Availability and Acknowledgment

The source code and more information are available at the Simplicissimus web page [17]. This work is sponsored in part by SGI, Mountain View, California.

References

- [1] S. Schupp, D. Gregor, D. Musser, S.-M. Liu, Library transformations, in: First IEEE Internat. Workshop on Source Code Analysis and Manipulation (SCAM 2001), Florence, Italy, IEEE, IEEE, 2001, pp. 109–121.
- [2] J. V. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, M. Tholburn, POOMA: A framework for scientific simulations on parallel architectures, in: G. V. Wilson, P. Lu (Eds.), *Parallel Programming using C++*, MIT Press, 1996, pp. 553–594.

- [3] T. Veldhuizen, Blitz++, <http://oonumerics.org/blitz>.
- [4] R. Wille, Restructuring lattice theory: An approach based on hierarchies of concepts, in: I. Rival (Ed.), *Ordered Sets*, Reidel, Dordrecht-Boston, 1982, pp. 445–470.
- [5] R. Wille, Concept lattices and conceptual knowledge systems, *Computers and Mathematics with Applications* 23 (1992) 493–522.
- [6] M. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, 1999.
- [7] Silicon Graphics Inc., *Standard Template Library Programmer’s Guide*, <http://www.sgi.com/tech/stl/> (1997).
- [8] S. Schupp, D. Gregor, B. Osman, D. Musser, J. Siek, L.-Q. Lee, A. Lumsdaine, Concept-based component libraries and optimizing compilers, Tech. rep., RPI Computer Science Department Technical Report 02-02 (2002).
- [9] D. Kapur, D. R. Musser, A. A. Stepanov, Tecton: A language for manipulating generic objects, in: J. Staunstrup (Ed.), *Proceedings of a Workshop on Program Specification*, Aarhus, Denmark, Vol. 134 of *Lecture Notes in Computer Science*, Springer-Verlag, 1981, pp. 402–414.
- [10] D. Musser, S. Schupp, R. Loos, Requirements-oriented programming, in: M. Jazayeri, R. Loos, D. Musser (Eds.), *Generic Programming—International Seminar*, Dagstuhl Castle, Germany 1998, *Selected Papers*, Vol. 1766 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2000, pp. 12–24.
- [11] T. Veldhuizen, Expression templates, in: *C++ Gems* [36].
- [12] CodeSourcery, LLC, G++ Internal Representation, <http://gcc.gnu.org/onlinedocs> (August 2000).
- [13] N. Myers, A new and useful template technique, in: *C++ Gems* [36].
- [14] ISO/IEC FDIS 14882, *International Standard for the C++ Programming Language*, American National Standards Institute (ANSI), X3 Secretariat, 1250 Eye Street NW, Suite 200, Washington, DC 20005, 1998.
- [15] K. Czarnecki, U. W. Eisenecker, *Generative Programming—Towards a New Paradigm of Software Engineering*, Addison Wesley Longman, 2000.
- [16] B. Dawes, D. Abrahams, Boost, <http://www.boost.org>.
- [17] *Simplicissimus*,
<http://www.cs.rpi.edu/research/gpg/Simplicissimus>.
- [18] J. R. Cordy, T. R. Dean, A. J. Malton, K. A. Schneider, Software engineering by source transformation—experience with TXL, in: *Proc. IEEE 1st International Workshop on Source Code Analysis and Manipulation*, Florence, November 2001, 2001, pp. 168–178.
- [19] J. R. Cordy, C. D. Halpern-Harnu, E. Promislow, TXL: A rapid prototyping system for programming language dialects, *Computer Languages* (1991) 97–107.

- [20] I. D. Baxter, C. W. Pidgeon, Software change through design maintenance, in: Proceedings of the IEEE Conference on Program Comprehension, Bari, 1997, pp. 250–259, citeseer.nj.nec.com/baxter97software.html.
- [21] M. Mehlich, I. Baxter, Mechanical tool support for high integrity software development, in: High Integrity Systems '97 (1997), IEEE Computer Society Press, Los Alamitos, California, USA., 1997, citeseer.nj.nec.com/mehlich97mechanical.html.
- [22] U. Aßmann, Graph Grammar Handbook, Chapman-Hall, 1998, Ch. OPTIMIX, A Tool for Rewriting and Optimizing Programs.
- [23] U. Aßmann, A. Ludwig, Aspect Weaving by Graph Rewriting, in: U. W. Eisenecker, K. Czarnecki (Eds.), Generative Component-based Software Engineering (GCSE), Vol. 1799 of Lecture Notes in Computer Science, Springer-Verlag, Erfurt, 1999.
- [24] D. Lacey, O. de Moor, Imperative program transformation by rewriting, in: R. Wilhelm (Ed.), International Conference on Compiler Construction, Springer Lecture Notes in Computer Science, 2001, pp. 52–68.
- [25] E. Visser, Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5, in: A. Middeldorp (Ed.), Rewriting Techniques and Applications (RTA'01), Vol. 2051 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 357–361.
- [26] O. Bagge, M. Haverlaan, E. Visser, CodeBoost: A framework for the transformation of C++ programs, Tech. rep., Universiteit Utrecht, The Netherlands (October 2000).
- [27] N. Jones, An introduction to partial evaluation, ACM Computing Surveys 28 (3) (1996) 480–503.
- [28] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: European Conference on Object-Oriented Programming (ECOOP'01), 2001.
- [29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwi, Aspect-oriented programming, in: European Conference on Object-Oriented Programming (ECOOP'97), 1997.
- [30] K. J. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston, 1996.
- [31] H. Ossher, M. Kaplan, W. Harrison, A. Katz, V. Kruskal, Subject-oriented composition rules, in: Proc. of the 1992 Conf. on Object-oriented Programming Systems, Languages and Applications, 1992.
- [32] C. Simonyi, The future is intentional, IEEE Computer .
- [33] S. Z. Guyer, C. Lin, An annotation language for optimizing software libraries, in: T. Ball (Ed.), 2nd Conference on Domain-Specific Languages, Usenix, 1999.

- [34] K. Davis, 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), Vol. 1543 of LNCS, Springer-Verlag, 1998.
- [35] D. R. Engler, Interface compilation: steps toward compiling programs, Transactions on Software Engineering 25 (3).
- [36] S. B. Lippman, C++ Gems, Cambridge University Press, 1996.

A The Rule SubscriptAtTo

```

namespace simplifier {
    using namespace expr_info;

    template<typename T>
    struct is_bounded_container
    {
        static const bool RET = false;
    };

    template<typename T, typename Alloc>
    struct is_bounded_container< std::vector<T, Alloc> >
    {
        static const bool RET = true;
    };

    struct SubscriptToAt
    {
        typedef StandardRuleClass rule_class;

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

        template<typename Cont, typename Index>
        struct Validator<
            Expr< BinaryExpr<Subscript, Cont, Index> > >
        {
            static const bool valid =
                is_bounded_container<typename Cont::result_type>::RET;
        };

        template<typename Expr> struct Result;
    };

```

```

    template<typename Cont, typename Index>
    struct Result<Expr<BinaryExpr<Subscript, Cont, Index> > >
    {
        typedef Expr< BinaryExpr<At, Cont, Index> > result;
    };
};
#define NEW_RULE SubscriptToAt
#include <simplifier/add_presimp_rule.h>
#undef NEW_RULE
}

```

B Trait Description of the Subscript-Operator

```

namespace expr_info {
    template<typename Value, typename Allocator>
    struct BinaryOpTraits<
        Subscript,
        std::vector<Value, Allocator>,
        typename std::vector<Value, Allocator>::size_type
    >
    {
        typedef std::vector<Value, Allocator> vector_type;
        typedef typename vector_type::size_type size_type;
        typedef __true_type is_applicative;
        typedef __false_type has_side_effects;
        typedef __true_type lhs_is_const;
        typedef __true_type rhs_is_const;
        typedef typename vector_type::reference result_type;
        static const int cost = SUBSCRIPT_COST;

        static inline result_type
        apply(vector_type& vec, const size_type& index)
        {
            return vec[index];
        }

        static inline typename vector_type::const_reference
        apply(const vector_type& vec, const size_type& index)
        {
            return vec[index];
        }
    };
}

```