International Journal of Parallel Programming, Vol. 33, No. 2, June 2005 (©2005)

Generic Programming and High-Performance Libraries (Preprint)

Douglas Gregor,[1] Jaakko Järvi,[2] Mayuresh Kulkarni,[3] Andrew Lumsdaine,[1] David Musser,[3] Sibylle Schupp[4]

[1] Open Systems Laboratory, Indiana University, Bloomington, IN. {dgregor,lums}@osl.iu.edu
[2] Department of Computer Science, Texas A&M University, College Station, TX. jarvi@cs.tamu.edu
[3] Computer Science Department, Rensselaer Polytechnic Institute. Troy, NY. {kulkam,musser}@cs.rpi.edu
[4] Dept. of Computing Science, Chalmers University of Technology, S-41296 Göteborg, Sweden. schupp@cs.chalmers.se

Generic programming is an especially attractive paradigm for developing libraries for high-performance computing because it simultaneously emphasizes generality and efficiency. In the generic programming approach, interfaces are based on sets of specified requirements on types, rather than on any particular types, allowing algorithms to inter-operate with any data types meeting the necessary requirements. These sets of requirements, known as *concepts*, can specify syntactic as well as semantic requirements. Besides providing a powerful means of describing interfaces to maximize software reuse, concepts provide a uniform mechanism for more closely coupling libraries with compilers and for effecting domain-specific library-based compiler extensions. To realize this goal however, programming languages and their associated tools must support concepts as first-class constructs. In this paper we advocate better syntactic and semantic support to make concepts first-class and present results demonstrating the kinds of improvements that are possible with static checking, compiler optimization, and algorithm correctness proofs for generic libraries based on concepts.

Keywords: Generic programming, software libraries, static analysis, high-level optimization, formal verification

1. INTRODUCTION

Software libraries are an important means of achieving software reuse and capturing domain-specific knowledge. Libraries are particularly important in high-performance computing because significant domain expertise—necessary to support the application area and necessary to support high performance—must be captured. Typically, software libraries are considered to be collections of functions and data types. In this paper, we propose a broader view of libraries and advocate the generic programming approach to library construction. With the generic programming approach, libraries can be much more tightly coupled to compilers, allowing libraries to provide highly-reusable data types and algorithms, but domain-specific optimizations and analyses as well.

Several aspects of generic programming make it particularly attractive for developing libraries for high-performance computing. Generic programming emphasizes finding the most general (or abstract) formulations of algorithms and then implementing efficient generic representations of them. Although these two features, generality and efficiency, are often considered to be opposing forces, generic algorithms are expected to be usable in as many situations as possible without sacrificing any performance at all.

A key aspect of generic programming is that generic algorithms are specified in terms of *abstract properties of types*, not in terms of particular types. Following the terminology of Stepanov and Austern, we adopt the term *concept* to mean the formalization of an abstraction as a set of requirements on a type (or on a set of types) [1]. These requirements may be *semantic* as well as *syntactic*. Concepts are central to generic programming and provide a unifying basis for providing maximally reusable algorithms and for effecting closer coupling between compilers and libraries, thus enabling domain-specific optimizations and analyses.

Although many languages have support for "generics," concepts are not true first-class entities in current programming languages. As a result, it is difficult to fully leverage the potential of generic programming in modern software construction. For example, the work in [2] describes serious scalability issues and other difficulties that arise when attempting to realize generic programming in languages that do not support the expression of even simple concepts (e.g., those including only syntactic requirements). Section 2 analyzes the expression of syntactic requirements for concepts and their use in library development.

In almost all programming languages and all uses of concepts in actual software development practice to date, *semantic* requirements have only appeared in externally and informally expressed concepts, such as in the SGI concept descriptions for the STL [1, 3], rather than in a machine-

checkable concept language. The main exceptions have been the tagging of certain operators with semantic attributes such as commutativity and associativity, and checking for their presence during instantiation; e.g., in the Axiom computer algebra system [4] or in very high level prototyping languages like Maude [5] (which does allow the expression of semantic equations within the language, but does not back them up with formal inference capabilities beyond their use as rewriting rules in symbolic executions). In Section 3 we discuss less limited forms of semantic constraint checking implemented in STLlint, a tool we developed for static checking of C++ programs that use the STL or other libraries in the same spirit [6, 7]. We further discuss even more general forms of semantic constraint checking that are feasible using formal proof-checking methods.

In addition to constraints on functionality, semantic concepts can include *performance constraints*. We have experimented extensively with expression and organization of such constraints in *algorithm concept taxonomies*. A major use of such taxonomies is to provide a well-developed standard to refer to while designing and implementing a generic algorithm library. We began by developing sequential algorithm concept taxonomies [8] for two fundamental problem domains, sequence algorithms from the STL and graph algorithms from BGL [9]. In these cases, useful performance constraints to place on the algorithms were already fairly well-understood at the level of asymptotic bounds, but making distinctions between some of the algorithms in these domains requires more precision; finding ways to express that precision so that the constraints can make useful distinctions has been a major focus of the work. With parallel and distributed algorithms, there are additional challenges in developing a library standard in terms of concept taxonomies, as we discuss in section 4.

2. SYNTACTIC CONCEPTS

A concept consists of four different kinds of requirements: associated types, function signatures, semantic constraints, and complexity guarantees. The *associated types* of a concept specify mappings from the modeling type to other collaborating types (such as the mapping from a container to the type of its elements). The *function signatures* specify the operations that must be implemented for the modeling type. Alternatively, these can be expressed as *valid expressions*, which specify operator and function invocations that must be supported by the modeling type or types. A *syntactic concept* consists of just associated types and function signatures, whereas a *semantic concept* also includes semantic constraints and complexity guarantees [10]. A concept may incorporate the requirements of another concept, in which case the first concept is said to *refine* the second. Types that meet the requirements of a concept are said to *model* the concept.

Syntactic concept requirements have traditionally been specified informally in documentation [9, 11, 12] due to the lack of proper language support. Although C++ has proven successful for generic programming despite its lack of language support, we have also applied the generic programming approach to several object-oriented languages, including Generic Java, C#, and Eiffel, and have reported notable difficulties [2]. These languages use subtyping to constrain type parameters. Even though subtype-based constraints may not to be ideal for generic programming, most of the difficulties we encountered originate from how languages define subtyping, rather than being inherent to subtype-based constraints. Current object-oriented languages could be extended to better support generic programming without drastic modifications to or departing significantly from the object-oriented paradigm. In particular, this section discusses how expressing associated types and constraints on them could be better supported, and describes extensions needed to support *concept-bounded polymorphism*, *constraint propagation*, and *multi-type concepts*.

2.1. Concept-bounded polymorphism

Generic programming has its roots in the higher-order programming style often used in functional programming languages [13]. Functions are generalized by type and function parameters. The higher-order style can express generic functions, but has the obvious disadvantage of requiring a large number of parameters for generic functions; each function that the implementation of a generic function relies on must be explicitly passed to the generic function. This style obtains genericity using only unconstrained parametric polymorphism.

The C++ template system [14] implements unconstrained parametric polymorphism and has been used extensively for generic programming [9, 12]. However, the lack of first-class support for concepts in the language has spawned various usability problems. First, the compiler is unable to verify that a data type passed to a generic (template) function models the concepts required by the algorithm. For syntactic concepts, passing a nonconforming data type usually results in lengthy error messages referring to the implementation of the generic function instead of the actual point of error at the function call [15]. Second, the compiler is unable to verify (even syntactically) that a generic function uses only the operations defined in the concepts it requires. Thus, errors in generic function implementations often go unnoticed until a user provides a data type meeting only the minimal stated requirements (but not the implicit requirements of the generic function). Finally, it is often desirable to select from several implementations of a function based solely on the concepts modeled by the arguments, a process we refer to as *concept-based overloading*. For

instance, when applying a sorting algorithm to a data structure, we must consider how the elements in the data structure are accessed: if they can only be accessed linearly (as with a linked list) we might select a default algorithm, but if they can be accessed efficiently via indexing (as with an array) we can apply the more-efficient quicksort algorithm.

Ad hoc techniques exist in C++ to address the above shortcomings, although all are incomplete. Concept checking [16,17] verifies syntactic concept conformance of arguments to generic functions, reducing the amount of irrelevant and misleading information in error messages, while post-processing of compiler error messages [15] eliminates redundant information and improves clarity. Concept archetypes [16, 17], on the other hand, are minimal syntactic models of concepts that can be passed to generic functions to verify that the generic functions do not require syntax not captured in a concept. Concept-based overloading has enjoyed more success in C++, both via the widely-used method of tag dispatching [12] and via arbitrary overloading [18]. While each of these techniques has proven useful, C++ still lacks a coherent system for expressing concept constraints on type parameters.

A rudimentary approach for expressing constraints on type parameters is the *where clause* mechanism, various forms of which can be found in CLU [19], Theta [20], and Ada [21]. A where clause lists function signatures in the declaration of a generic function. The listed functions must exist at each call site, and are implicitly passed into the generic function. This makes calls to generic functions less verbose. Where clauses do not, however, provide a way to group requirements into reusable entities, i.e., concepts.

Haskell *type classes* [22] provide constraint mechanisms that share much in common with concepts. Type classes contain function signatures, and optionally their default implementations. Type class constraints define the "context," the set of functions that can be used in a generic function. The functions in required contexts are implicitly passed into the generic function. Types must be explicitly declared to be *instances* of type classes. Thus, when using type classes to represent concepts, the modeling relation between types and concepts is by nominal conformance. Type classes provide a relatively direct representation for concepts. Type classes cannot, however, properly encapsulate associated types, as discussed in [2].

ML *signatures* are a structural constraint mechanism that can represent syntactic concepts. A signature describes the public interface of a module, or *structure* as it is called in ML. A signature declares which type names, values (functions), and nested structures must appear in a structure. A signature also defines a type for each value, and a signature for each nested structure. The ML mechanism for constrained genericity is *functors*, which

are metafunctions from structures to structures. Each argument of a functor is constrained to conform to a particular signature. This is less than ideal for generic programming, where one wants to constrain the type parameters of a single function. Each structure parameter to a functor must be passed in explicitly, which makes calls verbose [2].

2.2. Associated types

Associated type constraints are a mechanism to encapsulate constraints on several functionally dependent types into one entity. For example, consider Figures 1 and 2 showing two concepts from the domain of graphs. The Incidence Graph concept requires the existence of vertex and edge associated types, and places constraints on them.

Expression	Return Type or Description
Edge::vertex_type	Associated vertex type
source(e)	***Edge::vertex_type***
target(e)	***Edge::vertex_type***

Fig. 1. Graph Edge concept. Type ***Edge*** is a model of Graph Edge if the above requirements are satisfied. Object ***e*** is of type ***Edge***.

Expression	Return Type or Description
Graph::vertex_type	Associated vertex type
Graph::edge_type	Associated edge type
Graph::out_edge_iterator	Associated iterator type
out_edge_iterator::value_type == ***edge_type***	
edge_type models Graph Edge	
out_edge_iterator models Iterator	
out_edges(v,g)	***out_edge_iterator***
out_degree(v,g)	***out_edge_iterator***

Fig. 2. Incidence Graph concept. Type ***Graph*** is a model of Incidence Graph if the above requirements are satisfied. Object ***g*** is of type ***Graph*** and object ***v*** is of type ***Graph::vertex_type***.

All but the most trivial concepts have associated type requirements, and thus a language for generic programming must support their expression. Within C++, associated types (and other concept information) are typically encapsulated within traits classes [23]. Generic Java and C# do not,

however, provide a way to access and place constraints on type members of generic type parameters. Nevertheless, associated types can be emulated using other language mechanisms. A common idiom used to work around the lack of support for associated types is to add a new type parameter for each associated type. This approach is frequently used in practice. The C# ***IEnumerable<T>*** interface, from the Generic C# collection library, for iterating through containers serves as an example. When a type implements ***IEnumerable<T>*** it must bind a concrete value, the value type of the container, to the type parameter ***T***. The graph concepts in Figure 1 and 2 can be expressed as follows:

```
interface GraphEdge<Vertex> {
  Vertex source();
  Vertex target();
}
interface IncidenceGraph<Vertex, Edge, OutEdgeIter>
  where Edge : GraphEdge<Vertex>,
        OutEdgeIter : IEnumerable<Edge> {
  OutEdgeIter out_edges(Vertex v);
  int out_degree(Vertex v);
}
```

The main problem with this technique is that it fails to encapsulate associated types and their constraints into a single concept abstraction. Every use of a concept as a constraint of a generic function or a refinement declaration must list all of its associated types, and all constraints on those types. In a concept with several associated types, this becomes burdensome. In the study described in [2], the number of type parameters in generic algorithms was often more than doubled due to this effect.

Adding a direct representation for associated types to an object-oriented language, such as Generic C#, can be achieved by allowing *member types* in interfaces. Such members are placeholders for types, for which interfaces can place subtype constraints. Classes implementing such interfaces must bind a concrete value to every member type.

As an example, using member types the graph concepts from Figures 1 and 2 could be expressed as:

```
interface GraphEdge {
  type Vertex;
  Vertex source();
  Vertex target();
}
interface IncidenceGraph {
```

```
    type Vertex;
    type Edge : GraphEdge;
    Vertex == Edge.Vertex;

    type OutEdgeIter : IEnumerable<Edge>;

    OutEdgeIter out_edges(Vertex v);
    int out_degree(Vertex v);
}
```

The ***GraphEdge*** interface declares the member type ***Vertex*** and the ***IncidenceGraph*** interface has two associated types: ***Vertex*** and ***Edge***. Note the two constraints: ***Edge*** must be a subtype of ***GraphEdge***; and ***Vertex*** must be the same type as the associated type, also named ***Vertex***, of ***Edge***. The member types correspond directly to the associated types in Figure 2, and the subtype constraints correspond to requirements that types model concepts. A translation from the member type representation for associated types into the above described emulation that uses an extra type parameter for each associated type is described in [24].

2.3. Constraint propagation

Mainstream object-oriented languages do not support *constraint propagation*; the constraints on the type parameters to generic types do not automatically propagate to uses of those types. For example, although a container concept may require that its iterator type model a specified iterator concept, any generic algorithm using that container concept will still need to repeat the iterator constraint. As another example, consider the declaration of a function for finding the first neighbor of a vertex in a graph;

```
G_Vertex first_neighbor<G, G_Vertex,
    G_Edge, G_OutEdgeIter>(G g, G_Vertex v)
  where G : IncidenceGraph
              <G_Vertex, G_Edge, G_OutEdgeIter>;
```

Without constraint propagation, the declaration becomes:

```
G_Vertex first_neighbor<G, G_Vertex,
    G_Edge, G_OutEdgeIter>(G g, G_Vertex v)
  where G : IncidenceGraph
              <G_Vertex, G_Edge, G_OutEdgeIter>,
        G_Edge : GraphEdge<G_Vertex>,
        G_OutEdgeIter : IEnumerable<G_Edge>;
```

The additional constraints in this example merely repeat properties of the associated types of ***G*** which are already specified by the ***IncidenceGraph***

interface. A type cannot be bound to ***G*** unless it inherits ***IncidenceGraph***. This requires the type to provide the associated types ***Vertex***, ***Edge***, and ***OutEdgeIter***, which must satisfy the constraints specified in the ***IncidenceGraph*** interface. Thus, the compiler could safely assume that ***G_Vertex***, ***G_Edge***, and ***G_OutEdgeIter*** in the generic ***first_neighbor*** function also satisfy the constraints in ***IncidenceGraph***. Not making this assumption greatly increases the verbosity of generic code and adds extra dependencies on the exact contents of ***IncidenceGraph*** , thus breaking the encapsulation of the concept abstraction. This problem is not inherent to subtype-based constraint mechanisms. For example, the Cecil language automatically propagates constraints to uses of generic types [25, § 4.2]. Constraint propagation can be implemented by copying the type parameter constraints from each interface to each of the uses of the interface.

2.4. Constraining multiple types

Some abstractions define interactions between multiple independent types, in contrast to an abstraction with a main type and several associated types. An example of this is the mathematical concept Vector Space in Figure 3 (more examples can be found in [26]).

Expression	Return Type or Description
mult(v, s)	***V***
mult(s, v)	***V***

Fig. 3. Vector Space concept. Types ***V*** and ***S*** model the Vector Space concept if, in addition to the type ***S*** modeling the Field concept and the type ***V*** modeling the Additive Abelian Group concept, the above requirements are satisfied. Object ***v*** is of type ***V*** and object ***s*** is of type ***S***.

In this example it is tempting to think that the scalar type should be an associated type of the vector type. For example, the template class ***vector***<***complex***<***float***>> would have ***complex***<***float***> as its scalar type. However, in general, the scalar type of a vector space is not *determined* by the vector type. The popular linear algebra subroutine library LAPACK contains examples that demonstrate this. One such example is the CLACRM subroutine, which multiplies a complex matrix by a real matrix. The vector-scalar multiplications performed in this subroutine contain multiplications between ***complex***<***float***> and ***float***, which are significantly more efficient than converting the second argument to a complex number and performing complex multiplication [27]. Modeling the scalar type of a vector as an associated type would lead to this inefficient algorithm.

It is cumbersome to express multi-type concepts using object-oriented interfaces and subtype-based constraints. One must split the concept into multiple interfaces:

***interface VectorSpace_Vector*<*V, S*>**
: *AdditiveAbelianGroup*<*V*>
{ ***V mult(S);*** }

***interface VectorSpace_Scalar*<*V, S*> : *Field*<*S*>**
{ ***V mult(V);*** }

Algorithms that require the Vector Space concept must specify two constraints now instead of one. In general, if a concept hierarchy has height n, and places constraints on two types per concept, then the number of subtype constraints needed in an algorithm is 2^n, an exponential increase in the size of the requirement specification. The constraint propagation extension discussed in Section 2.3 ameliorates this problem; the exponential increase in the number of requirements can be avoided. However, the interface designer must still separate concepts in an arbitrary fashion. This could be overcome by an automatic translation of multi-type concepts into several interfaces.

3. SEMANTIC CONCEPTS WITH PERFORMANCE REQUIREMENTS

Semantic concepts extend syntactic concepts by formalizing the behavior that all models of the concept must exhibit. Conformance to a concept is therefore a (partial) specification of behavior that can be verified mechanically. We augment these semantic concepts with performance requirements that ensure predictable and efficient performance of algorithms. Semantic concepts provide a solid framework in which semantic requirements and axioms can be described, allowing compilers to verify and use concept modeling to improve the development and use of domain-specific libraries. Unlike user-defined optimizations for single data types, the ability for concepts to describe myriad data types that cross-cut various domains makes concepts especially attractive as a way to unify analyses and optimizations for built-in types with those for user-defined types. In addition, independently developed libraries can be safely composed and take mutual advantage of library-specified analyses and optimizations. In this section we discuss the use of semantic concepts with performance requirements for static analysis and checking, optimization, and algorithm correctness proof checking.

3.1. Static analysis

STLlint [6, 7] is a static checker for C++ programs that makes use of library-supplied semantic specifications (e.g., from the C++ standard template library). By analyzing the behavior of abstractions at a high level and ignoring the implementation of the abstractions, STLlint is able to detect errors in the use of libraries that could not be detected with traditional language-level checking. For example, STLlint allows one to extend the use of semantic properties beyond simple attribute tag checking to include static detection of range violations (e.g., dereferencing a past-the-end iterator), or missing properties such as the somewhat subtle "multi-pass" requirement imposed in the Forward Iterator concept. Central to the design of STLlint is the notion of abstraction via concept and data-type specifications, which permit STLlint to analyze and check programs at a very high level of abstraction.

```
vector<Student_info>
extract_fails(vector<Student_info>& students)
{
  vector<Student_info> fail;
  vector<Student_info>::iterator iter = students.begin(),
                                 end = students.end();
  while (iter != end) {
    if (fgrade(*iter)) {
        fail.push_back(*iter);
        iter = students.erase(iter);
    } else
        ++iter;
  }
  return fail;
}
```

Fig. 4. A misguided optimization of a routine that extracts and erases students with failing grades from the incoming data structure.

The example code in Figure. 4 was presented in an introductory C++ text book [28] to illustrate the dangers of *iterator invalidation*. Iterator invalidation occurs when an operation alters a data structure such that iterators referring to elements of that data structure can no longer be used safely. The invalidation behavior of operations varies greatly across domains, but the semantic iterator concept—including requirements pertaining to invalidation—cross-cuts various domains. STLlint permits static

checking of iterators by analyzing at the concept level, and is thereby able to uncover this error to produce a meaningful, high-level error message:

Warning: attempt to dereference a *singular iterator*
if (fgrade(*iter)) {

STLlint extends the notion of concept archetypes discussed in section 2.1 to *semantic* archetypes, which emulate the behavior of the most restrictive model of a particular concept. These concept archetypes are used by STLlint to check that generic functions do not require additional semantic guarantees beyond what is stated by the semantic concept itself. For instance, the STL ***max_element*** generic algorithm, which returns an iterator to the maximum element of a sequence, depends on the multi-pass property of Forward Iterators, which permits an algorithm to traverse the elements in a sequence multiple times. STLlint can detect the semantic errors resulting from mischaracterizing the concept requirements of ***max_element*** using a semantic archetype of an Input Iterator, which permits only one traversal of the sequence.

Though not the main emphasis of STLlint, it does incorporate specifications of refinement relations in an algorithm concept taxonomy. An algorithm thus declares the concept it models most specifically. Algorithm specification extensions are introduced via entry/exit handlers for a particular concept: entry handlers check preconditions and exit handlers check-/enforce postconditions. For example, *sorting* algorithms introduce a *sortedness* property that can be used in checking for proper use of algorithms that require it, such as binary search.

3.2. Optimization

A concept-based static analysis of a program provides information vital to optimization of that program. We have investigated two forms of concept-based optimization: concept-based rewriting and algorithm selection based on concepts.

The Simplicissimus optimizer [29] is an abstraction of the simplifier component in a compiler. While a traditional simplifier performs expression-level rewrites such as $x + 0 \rightarrow x$ when x is a built-in integer, Simplicissimus instead applies rewrite rules based on the concepts of the data types. For instance, the rewrite rule $x + 0 \rightarrow x$ is semantically valid when $(x, +)$ models the Monoid concept. Figure 5 illustrates two concept-based rewrite rules, including their concept requirements and several concrete instances. For instance, the right-identity rule $x + 0 \rightarrow x$ can be applied to many operations on built-in data types, such as floating-point multiplication and integer bitwise ***and*** along with user-defined operations on abstract data types

such as string concatenation and matrix multiplication. The right inverse rule for the Group concept (second row of Figure 5) illustrates additional optimizing rewrites. The two concept-based rules have several advantages over the ten specific instances we have given:

1. Additional instances can be generated from the two concept-based rules. Thus, while the list of instances is always incomplete, the concept-based rules encapsulate every data type that models the appropriate concepts, requiring no further user intervention.
2. The concept-based rules are directly related to and derivable from the axioms governing the Monoid and Group concepts.
3. Introduction of a new data type without concept-based rewrite rules requires revisiting existing rewrite rules to determine which rule may apply. However, since concept analysis is a necessary first step for use of a new data type with a generic algorithm, optimization via concept-based rewrite rules comes essentially "for free."

Rewrite	Requirements	Instances
$x + 0 \to x$	$(x, +)$ models Monoid	$i * 1 \to i$ $\quad f * 1.0 \to f$ $b \wedge \text{true} \to b$ $\quad i \,\&\, \text{0xFFFF} \to i$ $\mathit{concat}(s, \text{""}) \to s$ $\quad A \cdot I \to A$
$x + (-x) \to 0$	$(x, +, -)$ models Group	$i + (-i) \to 0$ $\quad f * (1.0/f) \to 1.0$ $r * r^{-1} \to \frac{1}{1}$ $\quad A \cdot A^{-1} \to I$

Fig. 5. Concept-based rewrite rules can be instantiated for many user-defined and built-in data types, reducing the number of rewrite rules required while increasing the scope and effectiveness of each rule.

While optimization can be performed for a single set of predetermined concepts, our experience with Simplicissimus has shown that the ability to extend the optimizer with user-defined rewrite rules is of paramount importance. These rules are often library specific, incorporating some degree of domain knowledge and often specializing general expressions to specific function calls. For instance, an arbitrary-precision floating point number ***f*** can be inverted via the expression ***1.0 / f***, but high-performance numerical libraries such as LiDIA [30] often provide a more-efficient ***Inverse()*** function. The author of LiDIA would therefore introduce the rewrite rule ***1.0 / f*** $\to$ ***f.Inverse()*** whenever ***f*** is a LiDIA data type. Specializing rewrite rules such as these can improve performance particularly when multiple generic libraries are used in conjunction [31].

Simplicissimus is limited to expression-level transformations based only on local information. STLlint, on the other hand, provides global analysis based on user- or library-defined concepts. STLlint's high-level static analysis can compute flow-sensitive properties of user-defined data types to suggest algorithmic optimizations. For instance, STLlint produces the following warning when given a program that first sorts a data structure and later attempts to perform a linear search through that data structure:

> Warning: potential optimization: the incoming sequence `[first, last)` is sorted, but will be searched linearly with this algorithm. Consider replacing this algorithm with one specialized for sorted sequences (e.g., ***lower_bound***):
>
> ***vector<int>::iterator i = find(v.begin(), v.end(), 42);***

STLlint only suggests optimizations: it does not have enough semantic information to verify or implement them. However, complete verification of the semantic constraints needed to enable optimization would permit high-level optimizations that improve the asymptotic performance, e.g., by transforming a linear search into a binary search.

3.3. Proving correctness of generic algorithms

The kinds of semantic checks and suggestions for optimizations performance by STLlint are achieved with specialized inference methods. We have begun experiments to show that it is feasible to extend the use of concepts in mainstream programming to include more general semantic requirements. For example, the aforementioned ***max_element*** algorithm also requires that the sequence element type have a comparison functor defined on it (either by an overloading of the $<$ operator or supplied through a functor passed to ***max_element***) and that it obey the axioms of the Strict Weak Order concept (see Figure 6). Presence or absence of a functor with a suitable signature can be detected in languages such as ML or Haskell through the use of signatures. This is possible even in C++, currently through the use of the Boost Concept-Checking Library [16, 32] but possibly in the future with concept constraints expressed within the language as proposed by Stroustrup and Dos Reis [33–35] to the C++ standards committee. In none of these cases, however, is there provision to check for satisfaction of the axioms.

To implement a general semantic checking capability we are taking advantage of recent advances in proof languages and proof-checking systems that permit development and use of proofs at a generic level. In such a system, proofs can themselves be generic components, in the sense that one can express a proof once and subsequently instantiate it many times to

Irreflexive	($\forall\ x$: domain, not ($x < x$))
Transitive	($\forall\ x, y, z$: domain, $x < y$ and $y < z$ implies $x < z$)
E-Definition	($\forall\ x, y$: domain, $E(x, y)$ = (not ($x < y$) and not ($y < x$)))
E-Transitive	($\forall\ x, y, z$: domain, $E(x, y)$ and $E(y, z)$ implies $E(x, z)$)

Fig. 6. Axioms of a Strict Weak Order concept. From these axioms two additional properties of E, symmetry and reflexivity, can be derived as theorems, showing that E is in fact an equivalence relation. These axioms are the minimal requirements on $<$ for correctness of many search or sorting-related algorithms, including STL's ***max_element***, ***binary_search***, ***sort***, etc. Although they are specified in the C++ standard [14], there is currently no requirement on compiler or library implementors for any kind of formal check for their satisfaction when instantiating generic algorithms like ***max_element***.

prove more specific cases, in much the same way as one does with generic algorithms.

This strategy enables a second key idea, which is to concentrate on the specification and use of semantic properties of generic library components, rather than broader classes of software. There are several advantages to such concentration of effort. One is the greater payoff for the (considerable) effort required to carry out proofs, by amortization over the many possible instances. Another is that we do not depend on acceptance and mastery of this technology by large number of programmers; it need only be carried out by the relatively small number of software designers and programmers involved in generic library development. Programmers who merely use the libraries do not need to be able to produce or to understand the proofs involved.

The proofs needed in semantic concept-checking are thus supplied by library component developers along with the specified concept requirements of the components. Therefore the language processor must only do proof checking, not proof search. As is well-known in the automated or interactive theorem proving research community, it is much more efficient to check a given proof than it is to search for an *a priori* unknown proof.

For this approach to work, the proof language and checking capability must itself support generalization and specialization in a natural and effective way. A key breakthrough in this area is K. Arkoudas's notion of a *Denotational Proof Language* (DPL) [36], which he has implemented in his Athena language and proof checker. DPL proofs can be written at a sufficiently abstract level that they can be instantiated to prove properties showing constraints are satisfied in many different instances, just as

generic algorithms can be instantiated many different ways to produce different useful concrete algorithms.

Proof checking in Athena The Athena language is really two distinct (but interwoven) languages: one for ordinary computation, and one for proof. The computation (or *expression*) language is similar to ML (but with Scheme-like syntax); in particular, it has first-class functions, in that they can be passed into, and returned from, other functions. Athena has proof language constructs similar to those for ordinary computation, including first-class *methods*, the analog of ordinary functions, whose purpose is to carry out proofs, updating the *assumption base*, an associative memory of propositions that have been asserted or proved in a proof session. The assumption base is fundamental to Athena's approach to deduction; all proof activity centers around it.

The proof language analog of *expression* is called a *deduction*. Like expressions, deductions are *executed*. Proper deductions (ones which correctly use primitive or programmed inference methods) produce theorems and add them to the assumption base; improper deductions result in an error condition.

Organizing axioms, proofs, and theorems for reuse in Athena An apparent drawback of the Athena language is its lack of of code organization capabilities above the level of functions or methods; i.e., module, class, package or namespace constructs commonly found in mainstream languages intended for development of large programs. Nor is there a type parameterization construct like generics or templates, making it appear that functions, methods, axioms, theorems, and proofs must be "concrete," that is, about specific functions and constants, rather than generic.

We have been able to show, however, that we can achieve both good organization and genericity without such additional constructs, by taking advantage of Athena's first-class functions and methods. We package up sets of axioms into functions, pass them around to other functions and methods that need them—and only to those functions and methods, so no others have to search through them or have name conflicts with them. Furthermore, we simulate type-parameterization simply by parameterizing functions and methods by functions that carry operator mappings. This approach is illustrated in the way we have already formalized—and used in proofs—numerous properties of ordering concepts (such as partial ordering, strict weak ordering, total ordering); algebraic concepts (such as monoid, group, ring, integral domain, field), and sequential computation concepts (such as container, iterator, range).

4. PARALLEL AND DISTRIBUTED ALGORITHM CONCEPTS

In most of the literature, the performance of parallel and distributed algorithms is typically indicated only in terms of asymptotic bounds on numbers of messages and time complexities, omitting other performance issues. For example, local computation at a node is rarely accounted for. However, mobile and sensor networks, where local computation is at a premium, are becoming increasingly common. Thus, when deciding between algorithms, a designer should be aware of how much local computation is involved. In addition to specifying requirements, concept descriptions can also organize and present detailed actual performance measurements. A comprehensive parallel and distributed algorithm concept taxonomy thus aids in our understanding of algorithms, helps in the design of new ones (based on situations where no known algorithms for a particular concept refinement exist), and helps a system designer to pick the correct algorithm for a particular application.

The distributed algorithms concept taxonomy we are developing [37] classifies algorithms on seven orthogonal dimensions: (1) Problem. This classifies the algorithms based on the problem that they solve. (2) Topology of the underlying network. Some algorithms are designed for specialized topologies, while others are for arbitrary topologies. Further refining this concept leads to some of the well known topologies like ring, completely connected graph, etc. (3) Tolerance to component failures. Some algorithms do not tolerate any failures while some can tolerate particular kinds of failures. Further refining this concept leads to Byzantine and non-Byzantine failures of nodes and links. (4) Method of information sharing between processes. We have thus far concentrated on message passing. (5) Strategy of the algorithm. Further refining this concept leads to well known paradigms like centralized control, distributed control, randomized, compositional, heart beat, probe echo, etc. (6) Timing properties required from the underlying network. Further refining this concept leads to synchronous, asynchronous, and partially-synchronous networks. (7) Process management. This classification accounts for static and dynamic process management capabilities and for algorithms that allow new nodes to join in dynamically as opposed to those that do not.

We have begun exploring the development of a parallel algorithms taxonomy, and a corresponding generic library based on the data-parallel programming paradigm. Data-parallel programming can achieve greater efficiency than what is possible with current automated parallelizing compilers that transform sequential programs into parallel executables. This is true also of programming directly with low-level concurrency and communication mechanisms, such as threads, processes, locks, semaphores,

and messages, but data-parallel programs can generally be expressed at a higher level of abstraction. The programmer still thinks and programs in parallel, but more abstractly, thus reducing the complexity of parallel programming. As an alternative to a full data-parallel programming language, our concept-based library approach leverages the capabilities of a mainstream base language (in our case, C++) while concentrating the desired new functionality into library modules. Moreover, this generic programming approach is infinitely extensible and is adaptable—by design—to the needs of particular application domains.

5. CONCLUSION

The generic programming approach to the development of high-performance domain-specific libraries focuses on isolating the core data type functionality and performance requirements into concepts. Concepts are important as documentation of the requirements of generic algorithms, but are inadequately supported in existing programming languages. We have described several features missing from mainstream object-oriented programming languages required for expressing syntactic concepts, including associated types and constraint propagation. We have further motivated the need for first-class concept support in languages by presenting concept-based static analyses, compiler optimizations, and proof techniques capable of improving the development and use of domain-specific libraries within mainstream programming languages.

Our future work will involve unifying the notions of syntactic, semantic, and performance requirements on concepts into a single, cohesive syntax for a mainstream programming language. The initial stage of development will involve constructing development tools—a compiler, static analysis framework, and optimization framework—for the concept syntax. Based on these development tools we will adapt our existing generic libraries [9, 12, 38] and also develop new libraries in new application areas to take full advantage of concept-aware compilation.

ACKNOWLEDGMENTS

This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment. The authors also thank Ronald Garcia for his helpful comments.

REFERENCES

1. M. H. Austern, *Generic Programming and the STL*, Professional computing series, Addison-Wesley (1999).

2. R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock, A Comparative Study of Language Support for Generic Programming, *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 115–134, ACM Press (October 2003).

3. SGI, Standard Template Library Programmer's Guide, http://www.sgi.com/tech/stl/ (1997).

4. R. D. Jenks and R. S. Sutor, *AXIOM: The Scientific Computation System*, Springer Verlag, New York (1992).

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, Maude: Specification and Programming in Rewriting Logic, *Theoretical Computer Science*, **285**:187–243 (2001), URL citeseer.nj.nec.com/clavel01maude.html.

6. D. Gregor and S. Schupp, Making the Usage of STL Safe, J. Gibbons and J. Jeuring (eds.), *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pp. 127–140, Kluwer, Boston (2002).

7. D. Gregor, *High-Level Static Analysis for Generic Libraries*, Ph.D. thesis, Rensselaer Polytechnic Institute (April 2004).

8. D. R. Musser, Algorithm Concepts, http://www.cs.rpi.edu/~musser/gp/algorithm-concepts (2003).

9. J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley (2002).

10. D. Kapur and D. Musser, *Tecton: a framework for specifying and verifying generic system components*, Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180 (July 1992), URL http://citeseer.ist.psu.edu/kapur92tecton.html.

11. Silicon Graphics, Inc., *SGI Implementation of the Standard Template Library* (2004), http://www.sgi.com/tech/stl/.

12. A. A. Stepanov and M. Lee, *The Standard Template Library*, Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project (May 1994).

13. A. Kershenbaum, D. Musser, and A. Stepanov, *Higher Order Imperative Programming*, Technical Report 88-10, Rensselaer Polytechnic Institute (1988), URL http://citeseer.ist.psu.edu/kershenbaum88higher.html.

14. International Standardization Organization (ISO), *ANSI/ISO Standard 14882, Programming Language C++*, 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland (1998).

15. L. Zolman, An STL Error Message Decryptor for Visual C++, *C/C++ Users Journal* (July 2001).

16. J. Siek and A. Lumsdaine, Concept Checking: Binding Parametric Polymorphism in C++, *First Workshop on C++ Template Programming* (October 2000), URL http://oonumerics.org/tmpw00/.

17. J. Willcock, J. Siek, and A. Lumsdaine, Caramel: A Concept Representation System for Generic Programming, *Second Workshop on C++ Template Programming* (October 2001), URL http://oonumerics.org/tmpw01/willcock.pdf.

18. J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine, Function Overloading Based on Arbitrary Properties of Types, *C/C++ Users Journal*, **21**(6):25–32 (June 2003).

19. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, Abstraction mechanisms in CLU, *Communications of the ACM*, **20**(8):564–576 (1977).

20. M. Day, R. Gruber, B. Liskov, and A. C. Myers, Subtypes vs. where clauses: Constraining parametric polymorphism, *OOPSLA*, pp. 156–158 (1995).

21. United States Department of Defense, *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edn. (February 1983).

22. P. Wadler and S. Blott, How to make ad-hoc polymorphism less ad-hoc, *ACM Symposium on Principles of Programming Languages*, pp. 60–76, ACM (January 1989), URL citeseer.nj.nec.com/wadler88how.html.

23. N. Myers, A new and useful technique: "traits", *C++ Report*, **7**(5):32–35 (June 1995).

24. J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock, An Analysis of Constrained Polymorphism for Generic Programming, K. Davis and J. Striegnitz (eds.), *Multiparadigm Programming in Object-Oriented Languages Workshop (MPOOL) at OOPSLA*, Anaheim, CA (October 2003).

25. C. Chambers and the Cecil Group, *The Cecil Language: Specification and Rationale, Version 3.1*, University of Washington, Computer Science and Engineering (December 2002), www.cs.washington.edu/research/projects/cecil/.

26. S. P. Jones, M. Jones, and E. Meijer, Type classes: an exploration of the design space, *Haskell Workshop* (June 1997), URL http://citeseer.ist.psu.edu/peytonjones97type.html.

27. X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, Design, Implementation and Testing of Extended and Mixed Precision BLAS, *ACM Transactions on Mathematical Software*, **28**(2):152–205 (June 2002).

28. A. Koenig and B. E. Moo, *Accelerated C++*, Addison-Wesley (2000).

29. S. Schupp, D. Gregor, D. Musser, and S.-M. Liu, Semantic and behaviorial library transformations, *Special Issue of the Journal of Information and Software Technology*, **44**(13):797–810 (October 2002).

30. The LiDIA Group, LiDIA—A C++ Library for Computational Number Theory, http://www.informatik.tu-darmstadt.de/TI/LiDIA/.

31. S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu, User-Extensible Simplification–Type-Based Optimizer Generators, *International Conference on Compiler Construction*, number 2027 in LNCS, pp. 86–101 (2001).

32. J. Siek, *Boost Concept Check Library*, Boost (2000), http://www.boost.org/libs/concept_check/.

33. B. Stroustrup and G. Dos Reis, *Concepts – Design choices for template argument checking*, Technical Report N1522=03-0105, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (October 2003), http://www.open-std.org/jtc1/sc22/wg21.

34. B. Stroustrup, *Concepts – A more abstract complement to type checking*, Technical Report N1510=03-0093, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (October 2003), http://www.open-std.org/jtc1/sc22/wg21.

35. B. Stroustrup and G. Dos Reis, *Concepts – syntax and composition*, Technical Report N1536=03-0119, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (October 2003), http://www.open-std.org/jtc1/sc22/wg21.

36. K. Arkoudas, *Denotational Proof Languages*, Ph.D. thesis, MIT (2000).

37. M. Kulkarni and D. R. Musser, Concept Taxonomy for Distributed Algorithms, http://www.cs.rpi.edu/~kulkam/concepts/pagedir/concindex.html.

38. J. G. Siek and A. Lumsdaine, The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra, *International Symposium on Computing in Object-Oriented Parallel Environments* (1998).