Generic Programming and High-Performance Libraries

Jaakko Järvi, Andrew Lumsdaine
Indiana University
{jajarvi,lums}@cs.indiana.edu

D. P. Gregor, M. Kulkarni, D. R. Musser, S. Schupp*
Rensselaer Polytechnic Institute
{gregod,kulkam,musser}@cs.rpi.edu, schupp@cs.chalmers.se

Abstract

Generic programming is an attractive paradigm for developing libraries for high-performance computing because of the simultaneous emphases placed on generality and efficiency. In this approach, interfaces are based on sets of specified requirements on types, rather than on any particular type, allowing algorithms to inter-operate with any data type meeting the necessary requirements. These sets of requirements, known as concepts, can specify syntactic as well as semantic requirements. Although concepts are fundamental to generic programming, they are not supported as first-class entities in mainstream programming languages, thus limiting the degree to which generic programming can be effectively applied. In this paper we advocate better syntactic and semantic support for concepts and describe some straightforward language features that could better support them. We also briefly discuss uses for concepts beyond their use in constraining polymorphism.

1 Introduction

Generic programming is an emerging programming paradigm for creating highly reusable domain-specific software libraries. Several aspects of this approach make it 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.

Fundamental to realizing generic algorithms is the notion of abstraction: 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) [2]. These requirements may be *semantic* as well as *syntactic*.

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 [12] 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 3 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 [2, 28], 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 [18] or in very high level prototyping languages like Maude [8] (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 4 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 [13, 14]. 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 [27] for two fundamental problem domains, sequence algorithms from the STL and graph

*Present address: Dept. of Computing Science, Chalmers University of Technology, S-41296 Göteborg, Sweden.

Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)
IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

algorithms from BGL [30]. 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 5.

2 Concept-bounded polymorphism

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 [21]. 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.

Generic programming has its roots in the higher-order programming style often used in functional programming languages [22]. 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.

A rudimentary approach for expressing constraints on type parameters is the *where clause* mechanism, various forms of which can be found in CLU [26], Theta [10], and Ada [35]. 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* [36] 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 [12].

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 [12].

2.1 Bounded quantification

To describe constrained polymorphism mechanisms we use the general setting of *qualified types* [19] to allow for a more uniform presentation. A qualified type is of the form $P => \tau$ where P is some predicate expression and τ is a type expression. The intuition is that if P is satisfied then $P => \tau$ has type τ. With $\overline{t}$ representing one or more type parameters, a qualified polymorphic type is written:

$$\forall \overline{t}.\ P => \tau$$

In this framework, a concept-bounded type is a qualified type where the predicates are assertions $c_i(\overline{t_i})$ stating that the types $\overline{t_i}$ model the concept c_i. Thus, concept-bounded polymorphic types have the form:

$$\forall \overline{t}.\ c_1(\overline{t_1}) \wedge \cdots \wedge c_n(\overline{t_n}) => \tau$$

where $\overline{t_i} \subseteq \overline{t}$ and τ is a type expression possibly referring to types in $\overline{t}$.

Cardelli and Wegner [6] were the first to suggest using subtyping to express constraints. The basic idea is to use subtyping assertions in the predicate of a qualified type. The predicates are of the form $t \leq \sigma$ where $\leq$ denotes the subtype relation, t is a type variable, and σ is a type expression. In this approach, polymorphic types are of the form

$$\forall t.\ t \leq \sigma => \tau[t]$$

where the type expression $\tau[t]$ may refer to t.

In the initial form of *bounded quantification*, the type expression σ is not allowed to refer to t. This restriction

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

was removed in the generalization to *F-bounded polymorphism* by Canning et al. [5], which allows recursive constraints. F-bounded polymorphism was further generalized to systems of mutually recursive subtyping constraints by Curtis [9, 11]. A recursively subtype-constrained type is of the form:

$$\forall \bar{t}.\ \tau_1 \leq \sigma_1 \wedge \cdots \wedge \tau_n \leq \sigma_n => \tau$$

where the type variables in $\bar{t}$ can appear anywhere in the type expressions τ_i, σ_i, and τ. This is kind of constraints for type expressions, with some minor restrictions, are used in the generics extensions of Java and C#.

3 Supporting syntactic concepts

We reported notable difficulties in following the generic programming approach with several object-oriented languages, including Generic Java, C#, and Eiffel [12]. All 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 *constraint propagation* and *multi-type concepts*.

3.1 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.

All but the most trivial concepts have associated type requirements, and thus a language for generic programming must support their expression. 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 [12], 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***. 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 [17].

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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

Figure 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***

Figure 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*.**

3.2 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 from the ***IncidenceGraph*** interface. This requires the type to provide the associated types ***Vertex***, ***Edge***, and ***OutEdgeIter***, such that they 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 the ***IncidenceGraph*** interface, 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 [7, § 4.2]. Constraint propagation can be implemented by copying the type parameter constraints from each interface to each of the uses of the interface.

3.3 Subclassing vs. subtyping

In subtype-bounded polymorphism constraints are imposed using the subtyping relation, so the expressiveness of the constraints very much depends on how the subtype relation is defined in the language. Much of the literature on bounded and F-bounded polymorphism [5,6] discusses languages with records, variants, and recursive types, and a structural subtyping relation. Main-stream object-oriented languages, however, use a subtype relation based on named conformance, which requires explicit subtype declarations in addition to the structural conformance requirement.

Object-oriented languages commonly unify the subtype and the subclass relation, which is established in the class declarations. This prevents later additions to the set of superclasses of a given class, directly affecting the modeling relation between types and concepts: If a type structurally conforms to the requirements of a generic algorithm, but is not a nominal subtype of the required interface, the type cannot be used as the type parameter of the algorithm; types cannot retroactively be declared to be models of a given concept. Such *retroactive modeling* is, however, important

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

when combining separately developed libraries.

The lack of retroactive modeling is not an inherent problem of subtype-based constraints. Retroactive subtyping can be implemented for object-oriented languages, as demonstrated by several authors [3,4,7,15,24]. Retroactive modeling is further discussed in [12].

3.4 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 [20]).

In this example it is tempting to think that the scalar type should be an associated type of the vector type. For example, the 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 [25]. Modeling the scalar type of a vector as an associated type would lead to this inefficient algorithm.

It is cumbersome to express multi-parameter 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 3.2 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-parameter concepts into several interfaces.

4 Semantic concept checking

We are gaining valuable experience with semantic concept checking in the forms in which it is implemented in STLlint [13, 14]. STLlint allows one to extend the use of semantic properties beyond 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 (e.g., the STL ***max_element*** generic algorithm, which returns an iterator to the maximum element of a sequence, depends on the multi-pass property, and STLlint can detect the error of applying ***max_element*** to a sequence accessed through ***istream_iterator***s, which belong only to the Input Iterator concept and not to Forward Iterator).

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, or to suggest an algorithm optimization, as illustrated by the following actual STLlint warning:

> 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.

These kinds of semantic checks and suggestions for optimizations are achieved in STLlint 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 4). 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 [29, 31] but possibly in the future with concept constraints expressed within the language as proposed by Stroustrup and Dos Reis [32–34] 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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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

Figure 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*.**

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 $E(x, z)$)

Figure 4. 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 [16], 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*.**

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 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) [1], 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's proof language has language constructs 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 lan-

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

guages 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).

5 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 [23] 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, while others 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.

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] K. Arkoudas. *Denotational Proof Languages*. PhD thesis, MIT, 2000.

[2] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[3] G. Baumgartner, M. Jansche, and K. Läufer. Half & Half: Multiple Dispatch and Retroactive Abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Ohio State University, 2002.

[4] G. Baumgartner and V. F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice and Experience*, 25(8):863–889, August 1995.

[5] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, 1989.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] C. Chambers and the Cecil Group. *The Cecil Language: Specification and Rationale, Version 3.1*. University of Washington, Computer Science and Engineering, Dec. 2002. www.cs.washington.edu/research/projects/cecil/.

[8] 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.

[9] P. Curtis. *Constrained quantification in polymorphic type analysis*. PhD thesis, Cornell University, Feb. 1990. www.parc.xerox.com/company/history/publications/bw-ps-gz/csl90-1.ps.gz.

[10] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA*, pages 156–158, 1995.

[11] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1. Elsevier, 1995.

[12] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134. ACM Press, Oct. 2003.

[13] D. Gregor. *Static Analysis of Generic Component Composition*. PhD thesis, Rensselaer Polytechnic Institute, forthcoming.

[14] D. Gregor and S. Schupp. Making the usage of STL safe. In J. Gibbons and J. Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 127–140, Boston, 2003. Kluwer.

[15] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 36–55. Springer, July 1993.

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

[17] J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An analysis of constrained polymorphism for generic programming. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming in Object-Oriented Languages Workshop (MPOOL) at OOPSLA*, Anaheim, CA, Oct. 2003.

[18] R. D. Jenks and R. S. Sutor. *AXIOM: The Scientific Computation System*. Springer Verlag, New York, 1992.

[19] M. P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[20] S. P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[21] 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.

[22] A. Kershenbaum, D. Musser, and A. Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, 1988.

[23] M. Kulkarni and D. R. Musser. Concept taxonomy for distributed algorithms. http://www.cs.rpi.edu/~kulkam/concepts/pagedir/concindex.html.

[24] K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for Java. *Computer Journal*, 43(6):469–481, 2001.

[25] 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.

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

[27] D. R. Musser. Algorithm concepts. http://www.cs.rpi.edu/~musser/gp/algorithm-concepts, 2003.

[28] SGI. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/, 1997.

[29] J. Siek. *Boost Concept Check Library*. Boost, 2000. http://www.boost.org/libs/concept_check/.

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

[31] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[32] 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++, Oct. 2003. http://anubis.dkuug.dk/jtc1/sc22/wg21.

[33] B. Stroustrup and G. D. 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++, Oct. 2003. http://anubis.dkuug.dk/jtc1/sc22/wg21.

[34] B. Stroustrup and G. D. Reis. Concepts — syntax and composition. Technical Report N1536=03-0119, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Oct. 2003. http://anubis.dkuug.dk/jtc1/sc22/wg21.

[35] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983.

[36] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.