# A Comparative Study of Language Support for Generic Programming

Ronald Garcia Jaakko Järvi Andrew Lumsdaine
Jeremy Siek Jeremiah Willcock
Open Systems Lab
Indiana University Bloomington
Bloomington, IN USA
{garcia,jajarvi,lums,jsiek,jewillco}@osl.iu.edu

## ABSTRACT

Many modern programming languages support basic generic programming, sufficient to implement type-safe polymorphic containers. Some languages have moved beyond this basic support to a broader, more powerful interpretation of generic programming, and their extensions have proven valuable in practice. This paper reports on a comprehensive comparison of generics in six programming languages: C++, Standard ML, Haskell, Eiffel, Java (with its proposed generics extension), and Generic C#. By implementing a substantial example in each of these languages, we identify eight language features that support this broader view of generic programming. We find these features are necessary to avoid awkward designs, poor maintainability, unnecessary run-time checks, and painfully verbose code. As languages increasingly support generics, it is important that language designers understand the features necessary to provide powerful generics and that their absence causes serious difficulties for programmers.

## Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*; D.3.2 [**Programming Languages**]: Language Classifications—*multiparadigm languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, constraints, polymorphism*

## General Terms

Languages, Design, Standardization

## Keywords

generics, generic programming, polymorphism, C++, Standard ML, Haskell, Eiffel, Java, C#

## 1. INTRODUCTION

Generic programming is an increasingly popular and important paradigm for software development and many modern programming languages provide basic support for it. For example, the use of type-safe polymorphic containers is routine programming practice today. Some languages have moved beyond elementary generics to a broader, more powerful interpretation, and their extensions have proven valuable in practice. One domain where generic programming has been particularly effective is reusable libraries of software components, an example of which is the Standard Template Library (STL), now part of the C++ Standard Library [23,45]. As the generic programming paradigm gains momentum, it is important to clearly and deeply understand the language issues. In particular, it is important to understand what language features are required to support the broader notion of generic programming.

To aid in this process, we present results of an in-depth study comparing six programming languages that support generics: Standard ML [35], C++ [17], Haskell [21], Eiffel [30], Java (with the proposed genericity extension) [6], and Generic C# [24,33]. The first four currently support generics while the latter two have proposed extensions (and prototype implementations) that do so. These languages were selected because they are widely used and represent the state of the art in programming languages with generics.

Our high-level goals for this study were the following:

- Understand what language features are necessary to support generic programming;
- Understand the extent to which specific languages support generic programming;
- Provide guidance for development of language support for generics; and
- Illuminate for the community some of the power and subtleties of generic programming.

It is decidedly *not* a goal of this paper to demonstrate that one language is "better" than any other language. This paper is also not a comparison of generic programming to object-oriented programming (or to any other paradigm).

To conduct the study, we designed a model library by extracting a small but significant example of generic programming from a state-of-the art generic library (the Boost Graph Library [41]). The model library was fully implemented in all six target languages. This example was chosen because it includes a variety of generic programming techniques (some beyond the scope of, say, the STL) and could therefore expose many subtleties of generic programming. We attempted to create a uniform implementation across all of the languages while still using the standard techniques and idioms of each language. For each implementation, we evaluated the language features available to realize different facets of generic programming. In addition, we evaluated each implementation with respect to software quality issues that generic programming enables, such as modularity, safety, and conciseness of expression.

The results of this process constitute the main results of this paper and are summarized in Table 1. The table lists the eight language features that we identified as being important to generic programming and shows the level of support for that feature in each language. We find these features are necessary for the development of high-quality generic libraries. Incomplete support of these features can result in awkward designs, poor maintainability, unnecessary run-time checks, and painfully verbose code. As languages increasingly support generics, it is important that language designers understand the features necessary to provide powerful generics and that their absence causes serious difficulties for programmers.

The rest of this paper describes how we reached the conclusions in the table and why those language properties are important. The paper is organized as follows. Section 2 provides a brief introduction to generic programming and defines the terminology we use in the paper. Section 3 describes the design of the generic graph library that forms the basis for our comparisons. Sections 4 through 9 present the individual implementations of the graph library in the selected languages. Each of these sections also evaluates the level of support for generic programming provided by each language. In Section 10 we discuss in detail the most important issues we encountered during the course of this study and provide a detailed explanation of Table 1. We present some conclusions in Section 11.

## 2. GENERIC PROGRAMMING

Definitions of generic programming vary. Typically, generic programming involves type parameters for data types and functions. While it is true that type parameters are required for generic programming, there is much more to generic programming than just type parameters. Inspired by the STL, we take a broader view of generic programming and use the definition from [18] reproduced in Figure 1.

Associated with this definition, terminology and techniques for carrying out generic programming (and for supporting these key ideas) have emerged.

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

Figure 1: Definition of Generic Programming

### Terminology

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) [1]. These requirements may be semantic as well as syntactic. 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. Note that it is not necessarily the case that a concept will specify the requirements of just one type—it is sometimes the case that a concept will involve multiple types and specify their relationships.

Concepts play an important role in specifying generic algorithms. Since a concept may be modeled by any concrete type meeting its requirements, algorithms specified in terms of concepts must be able to be used with multiple types. Thus, generic algorithms must be polymorphic. For languages that explicitly support concepts, concepts are used to ***constrain*** type parameters.

Traditionally, a concept consists of ***associated types***, ***valid expressions***, semantic invariants, and complexity guarantees. The associated types of a concept specify mappings from the modeling type to other collaborating types (see Figure 4 for an example). Valid expressions specify the operations that must be implemented for the modeling type. At this point in the state of the art, type systems typically do not include semantic invariants and complexity guarantees. Therefore, we state that for a type to properly model a concept, the associated types and valid expressions specified by the concept must be defined.

These primary aspects of generic programming, i.e., generic algorithms, concepts, refinement, modeling, and constraints, are realized in different ways in our different target programming languages. The specific language features that are used to support generic programming are summarized in Table 2.

### Example

A simple example illustrates these generic programming issues. The example is initially presented in C++; Figure 2 shows versions in all six languages.

In C++, type parameterization of functions is accomplished with templates. The following is an example of a generic algorithm, realized as a ***function template*** in C++:

```
template <class T>
const T& pick(const T& x, const T& y) {
  if (better(x, y)) return x; else return y;
}
```

This algorithm applies the ***better*** function to its arguments and returns the first argument if ***better*** returns true, otherwise it returns the second argument.

Not every type can be used with ***pick***. The concept Comparable is defined to represent types that may be used with ***pick***. Unfortu-

| | C++ | Standard ML | Haskell | Eiffel | Java Generics | Generic C# |
|---|---|---|---|---|---|---|
| Multi-type concepts | - | ● | ●* | ○ | ○ | ○ |
| Multiple constraints | - | ◒ | ● | ○† | ● | ○‡ |
| Associated type access | ● | ● | ◒ | ◒ | ◒ | ◒ |
| Retroactive modeling | - | ● | ● | ○ | ○ | ○ |
| Type aliases | ● | ● | ● | ○ | ○ | ○ |
| Separate compilation | ○ | ● | ● | ● | ● | ● |
| Implicit instantiation | ● | ○ | ● | ○ | ● | ○‡ |
| Concise syntax | ● | ◒ | ● | ○ | ◒ | ○ |

*Using the multi-parameter type class extension to Haskell 98 [22]. †Planned language additions. ‡Planned for inclusion in Whidbey release of C#.

Table 1: The level of support for important properties for generic programming in the evaluated languages. "Multi-type concepts" indicates whether multiple types cane be simultaneously constrained. "Multiple constraints" indicates whether more than one constraint can be placed on a type parameter. "Associated type access" rates the ease in which types can be mapped to other types within the context of a generic function. "Retroactive modeling" indicates the ability to add new modeling relationships after a type has been defined. "Type aliases" indicates whether a mechanism for creating shorter names for types is provided. "Separate compilation" indicates whether generic functions are type-checked and compiled independently from their use. "Implicit instantiation" indicates that type parameters can be deduced without requiring explicit syntax for instantiation. "Concise syntax" indicates whether the syntax required to compose layers of generic components is independent of the scale of composition. The rating of "-" in the C++ column indicates that while C++ does not explicitly support the feature, one can still program as if the feature were supported due to the flexibility of C++ templates.

| Role | C++ | ML | Haskell | Eiffel | Java generics | Generic C# |
|---|---|---|---|---|---|---|
| Generic algorithm | function template | functor | polymorphic function | generic class | generic method | generic method |
| Concept | documentation | signature | type class | deferred class | interface | interface |
| Refinement | documentation | include | inheritance (⇒) | inherit | extends | inherit (:) |
| Modeling | documentation | implicit | instance | inherit | implements | inherit (:) |
| Constraint | documentation | param sig (:) | context (⇒) | conformance (→) | extends | where |

Table 2: The roles of language features used for generic programming.

nately, C++ does not support concepts directly so naming and documentation conventions have been established to represent them [1]. The Comparable concept is documented this way in C++:

| Comparable |
|---|
| ***bool better(const T&, const T&)*** |

Any type ***T*** is a model of Comparable if there is a ***better*** function with the given signature. For ***int*** to model Comparable, we simply define a ***better*** function for ***ints***:

```
bool better(int i, int j) { return j < i; }
```

In C++ it is customary to identify concepts by appropriately naming template parameters. The previous example would normally be written

```
template <class Comparable>
const Comparable&
pick(const Comparable& x, const Comparable& y) {
  if (better(x, y)) return x; else return y;
}
```

We define two types, ***Apple*** and ***Orange***

```
struct Apple {
  Apple(int r) : rating(r) {}
  int rating;
};
bool better(const Apple& a, const Apple& b)
  { return b.rating < a.rating; }

struct Orange {
  Orange(const string& s) : name(s) { }
  string name;
};
bool better(const Orange& a, const Orange& b)
  { return lexicographical_compare(b.name.begin(), b.name.end(),
                                   a.name.begin(), a.name.end()); }
```

***Apple*** and ***Orange*** model the Comparable concept implicitly via the existence of the ***better*** function for those types.

We finish by calling the generic algorithm ***pick*** with arguments of type ***int***, ***Apple***, and ***Orange***.

```
int main(int, char*[]) {
  int i = 0, j = 2;
  Apple a1(3), a2(5);
  Orange o1("Miller"), o2("Portokalos");

  int k = pick(i, j);
  Apple a3 = pick(a1, a2);
  Orange o3 = pick(o1, o2);

  return EXIT_SUCCESS;
}
```

## 3. A GENERIC GRAPH LIBRARY

To evaluate support for generic programming, a library of graph data structures was implemented in each language. The library provides generic algorithms associated with breadth-first search, including Dijkstra's single-source shortest paths and Prim's minimum spanning tree algorithms [13,39]. The design presented here descends from the generic graph library presented in [43], which evolved into the Boost Graph Library (BGL) [41].

Figure 3 depicts the graph algorithms, their relationships, and how they are parameterized. Each large box represents an algorithm and the attached small boxes represent type parameters. An arrow from one algorithm to another specifies that one algorithm is

C++

```
// concept Comparable:
// bool better(const T&, const T&)

template <class Comparable>
const Comparable& pick(const Comparable& x,
                       const Comparable& y) {
  if (better(x, y)) return x; else return y;
}

struct Apple {
  Apple(int r) : rating(r) {}
  int rating;
};
bool better(const Apple& a, const Apple& b)
  { return b.rating < a.rating; }

int main(int, char∗[]) {
  Apple a1(3), a2(5);
  Apple a3 = pick(a1, a2);
}
```

ML

```
signature Comparable =
sig
  type value_t
  val better : value_t ∗ value_t → bool
end

functor MakePick(C : Comparable) =
struct
  type value_t = C.value_t
  fun pick x y = if C.better(x,y) then x else y
end

structure Apple =
struct
  datatype value_t = AppleT of int
  fun create n = AppleT n
  fun better ((AppleT x),(AppleT y)) = y < x
end

structure PickApples = MakePick(Apple)
val a1 = Apple.create 5 and a2 = Apple.create 3
val a3 = PickApples.pick a1 a2
```

Haskell

```
class Comparable t where
  better :: (t, t) → Bool

pick :: Comparable t ⇒ (t, t) → t
pick (x, y) = if (better (x, y)) then x else y

data Apple = MkApple Int

instance Comparable Apple where
  better = (λ (MkApple m, MkApple n) → n < m)

a1 = MkApple 3; a2 = MkApple 5
a3 = pick(a1, a2)
```

Eiffel

```
deferred class COMPARABLE[T]
feature
  better (a: T) : BOOLEAN is deferred end
end

class PICK[T→ COMPARABLE[T]]
feature
  go (a: T; b: T) : T is do
    if a.better(b) then
      Result := a
    else
      Result := b
    end
  end
end

class APPLE inherit COMPARABLE[APPLE] end
create make
feature
  make(r: INTEGER) is do rating := r end
  better (a: APPLE) : BOOLEAN is do
    Result := rating < a.rating;
  end
feature {APPLE}
  rating : INTEGER
end

class ROOT_CLASS
create make
feature make is
  local
    a1, a2, a3 : APPLE;
    picker: pick[APPLE];
  do
    create picker;
    create a1.make(3); create a2.make(5);
    a3 := picker.go(a1, a2);
  end
end
```

Java Generics

```
interface Comparable<T> {
  boolean better(T x);
}

class pick {
  static <T extends Comparable<T>>
  T pick(T a, T b) {
    if (a.better(b)) return a; else return b;
  }
}

class Apple implements Comparable<Apple> {
  Apple(int r) { rating = r; }
  public boolean better(Apple x)
    { return x.rating < rating;}
  int rating;
}

public class Main {
  public static void main(String[] args) {
    Apple a1 = new Apple(3),
          a2 = new Apple(5);
    Apple a3 = pick.go(a1, a2);
  }
}
```

Generic C#

```
interface Comparable<T> {
  bool better(T x);
}

class pick {
  static T go<T>(T a, T b)
      where T : Comparable<T> {
    if (a.better(b)) return a; else return b;
  }
}

class Apple : Comparable<Apple> {
  public Apple(int r) {rating = r;}
  public bool better(Apple x)
    { return x.rating < rating; }
  private int rating;
}

public class Main_eg {
  public static int Main(string[] args) {
    Apple a1 = new Apple(3),
          a2 = new Apple(5);
    Apple a3 = pick.go<Apple>(a1,a2);
    return 0;
  }
}
```

Figure 2: Comparing ***Apples*** to ***Apples***. The Comparable concept, ***pick*** function, and ***Apple*** data type are implemented in each of our target languages. A simple example using each language is also shown.

implemented using the other. An arrow from a type parameter to an unboxed name specifies that the type parameter must model that concept. For example, the breadth-first search algorithm has three type parameters: ***G***, ***C***, and ***Vis***. Each of these have requirements: ***G*** must model the Vertex List Graph and Incidence Graph concepts, ***C*** must model the Read/Write Map concept, and ***Vis*** must model the BFS Visitor concept. Finally, breadth-first search is implemented using the graph search algorithm.

The core algorithm of this library is graph search, which traverses a graph and performs user-defined operations at certain points in the search. The order in which vertices are visited is controlled by a type argument, ***B***, that models the Bag concept. This concept abstracts a data structure with insert and remove operations but no requirements on the order in which items are removed. When ***B*** is bound to a FIFO queue, the traversal order is breadth-first. When it is bound to a priority queue based on distance to a source vertex, the order is closest-first, as in Dijkstra's single-source shortest paths algorithm. Graph search is also parameterized on actions to take at event points during the search, such as when a vertex is first discovered. This parameter, ***Vis***, must model the Visitor concept. The graph search algorithm also takes a type parameter ***C*** for mapping each vertex to its color and ***C*** is required to model the Read/Write Map concept.

The Read Map and Read/Write Map concepts represent variants of an important abstraction in the graph library: the ***property map***. In practice, graphs represent domain-specific entities. For example, a graph might depict the layout of a communication network, vertices representing endpoints and edges representing direct links. In addition to the number of vertices and the edges between them, a graph may associate values to its elements. Each vertex of a communication network graph might have a name and each edge a maximum transmission rate. Some algorithms require access to domain information associated with the graph representation. For example, Prim's minimum spanning tree algorithm requires "weight" information associated with each edge in a graph. Property maps provide a convenient implementation-agnostic means of expressing, to algorithms, relations between graph elements and domain-specific data. Some graph data structures directly contain associated values with each node; others use external associative data structures to express these relationships. Interfaces based on property maps work equally well with both representations.

The graph algorithms are all parameterized on the graph type. Graph search takes one type parameter ***G***, which must model two concepts, Incidence Graph and Vertex List Graph. The Incidence Graph concept defines an interface for accessing out-edges of a vertex. Vertex List Graph specifies an interface for accessing the vertices of a graph in an unspecified order. The Bellman-Ford shortest paths algorithm [4] requires a model of the Edge List Graph concept, which provides access to all the edges of a graph.

That graph capabilities are partitioned among three concepts illustrates generic programming's emphasis on algorithm requirements. The Bellman-Ford shortest paths algorithm requires of a graph only the operations described by the Edge List Graph concept. Graph search, in contrast, requires the functionality of both its required concepts. By partitioning the functionality of graphs, each algorithm can be used with any data type that meets its minimum requirements. If the three graph concepts were replaced with one, each algorithm would require more from its graph type parameter than necessary—and would thus unnecessarily restrict the set of types with which it could be used.

The graph library design is suitable for evaluating generic programming capabilities of languages because it includes a rich variety of generic programming techniques. Most of the algorithms are implemented using other library algorithms: breadth-first search and Dijkstra's shortest paths use graph search, Prim's minimum spanning tree algorithm uses Dijkstra's algorithm, and Johnson's all-pairs shortest paths algorithm uses both Dijkstra's and Bellman-Ford shortest paths. Type parameters for some algorithms, such as the ***G*** parameter to breadth-first search, must model multiple concepts. In addition, the algorithms require certain relationships between type parameters. For example, consider the graph search algorithm. The ***C*** type argument, as a model of Read/Write Map, is required to have an associated key type. The ***G*** type argument is required to have an associated vertex type. Graph search requires that these two types be the same.

The graph library is used throughout the remainder of this paper as a common basis for discussion. Though the entire library was implemented in each language, discussion is limited for brevity. We focus on the interface of the breadth-first search algorithm and the infrastructure surrounding it, including concept definitions and an example use of the algorithm. The interested reader can find the full implementations for each language, including instructions for compilation, at the following URL:
***http://www.osl.iu.edu/research/comparing/***

## 4. GRAPH LIBRARY IN C++

C++ generics were intentionally designed to exceed what is required to implement containers. The resulting template system provides a platform for experimentation with, and insight into the expressive power of, generic programming. Before templates, C++ was primarily considered an object-oriented programming language. Templates were added to C++ for the same reason that generics were added to several other languages in our study: to provide a means for developing type safe containers [46, §15.2]. Greater emphasis was placed on clean and consistent design than restriction and policy. For example, although function templates are not necessary to develop type-safe polymorphic containers, C++ has always supported classes and standalone functions equally; supporting function templates in addition to class templates preserves that design philosophy. Early experiments in developing generic functions suggested that more comprehensive facilities would be beneficial. These experiments also inspired design decisions that differ from the object-oriented generics designs (Java generics, Generic C#, and Eiffel). For example, C++ does not contain any explicit mechanism for constraining template parameters. During C++ standardization, several mechanisms were proposed for constraining template parameters, including subtype-based constraints. All proposed mechanisms were found to either undermine the expressive power of generics or to inadequately express the variety of constraints used in practice [46, §15.4].

Two C++ language features combine to enable generic programming: templates and function overloading. C++ includes both function templates and class templates; we use function templates to represent generic algorithms. We discuss the role of function overloading in the next section. In C++, templates are not separately type checked. Instead, type checking is performed after instantiation at each call site. Type checking of the bound types can only succeed when the input types have satisfied the type requirements of the function template body. Unfortunately, because of this, if a generic algorithm is invoked with an improper type, byzantine and potentially misleading error messages may result.

### 4.1 Implementation

The ***breadth_first_search*** function template is shown in Figure 4. C++ does not provide direct support for constraining type parameters; standard practice is to express constraints in documentation in

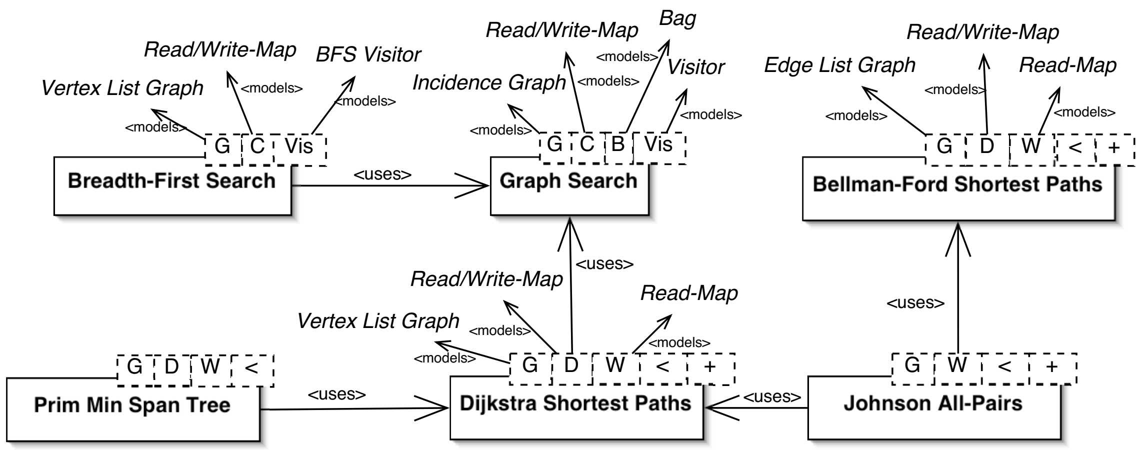

Figure 3: Graph algorithm parameterization and reuse within the graph library. Arrows for redundant models relationships are not shown. For example, the type parameter ***G*** of breadth-first search must also model Incidence Graph because breadth-first search uses graph search.

```
template <class G, class C, class Vis>
void breadth_first_search(const G& g,
   typename graph_traits<G>::vertex s, C c, Vis vis);
```

constraints:
- ***G*** models Vertex List Graph and Incidence Graph
- ***C*** models Read/Write Map
- ***map_traits<C>::key == vertex***
- ***map_traits<C>::value*** models Color
- ***Vis*** models BFS Visitor

Figure 4: Breadth-first search as a function template.

conjunction with meaningful template parameter names [1]. Techniques for checking constraints in C++ can be implemented as a library [29, 42]. These techniques, however, are distinct from actual language support and involve insertion of what are essentially compile-time assertions into the bodies of generic algorithms.

The ***graph_traits*** class template provides access to the associated types of the graph type. Here we use ***graph_traits*** to access the vertex type. Traits classes are an idiom used in C++ to map types to other types or functions [37]. A traits class is a class template. For each type in the domain of the map a specialized version of the class template is created containing nested typedefs and member functions. In Figure 5 we specialize ***graph_traits*** for the ***AdjacencyList*** class, which models ***Graph***.

Inside the ***breadth_first_search*** function, calls to functions associated with the concepts, such as ***out_edges*** from Incidence Graph, are resolved by the usual function overloading rules for C++. That is, each is resolved to the best overload for the given argument types.

Documentation for the graph concepts is shown in Table 3. In addition to function signatures, the concepts specify access to associated types such as vertex, edge, and iterator types through the ***graph_traits*** class.

A sketch of a concrete adjacency list implementation is shown in Figure 5. The ***AdjacencyList*** class is a model of the Incidence Graph and Vertex List Graph concepts, but this fact is implicit. There is no mechanism for specifying that ***AdjacencyList*** models these concepts. The ***graph_traits*** class is specialized for ***AdjacencyList*** so the associated types can be accessed from within function templates.

The definitions of the Read/Write Map and Read Map concepts are in Table 4 and the definition of the BFS Visitor concept is in Table 5.

| Graph |
|---|
| ***graph_traits<G>::vertex*** |
| ***graph_traits<G>::edge*** |
| ***vertex src(edge, const G&);*** |
| ***vertex tgt(edge, const G&);*** |
| **Incidence Graph refines Graph** |
| ***graph_traits<G>::out_edge_iter*** models Iterator |
| ***pair<out_edge_iter> out_edges(vertex, const G&);*** |
| ***int out_degree(vertex, const G&);*** |
| **Vertex List Graph** |
| ***graph_traits<G>::vertex_iter*** models Iterator |
| ***pair<vertex_iter> vertices(const G&);*** |
| ***int num_vertices(const G&);*** |

Table 3: Documentation for the graph concepts.

```
class AdjacencyList {
public:
  ...
private:
  vector< list<int> > adj_lists;
};
template <> struct graph_traits<AdjacencyList> {
  typedef int vertex;
  typedef pair<int, int> edge;
  typedef list<int>::const_iterator out_edge_iter;
  ...
};
```

Figure 5: Sketch of a concrete graph implementation.

| Read Map |
|---|
| ***map_traits<M>::key*** |
| ***map_traits<M>::value*** |
| ***value get(const M&, key);*** |
| **Read/Write Map refines Read Map** |
| ***void put(M&, key, value);*** |

Table 4: Documentation for the mapping concepts.

| BFS Visitor |
|---|
| *void V::discover_vertex(vertex, G);* |
| *void V::finish_vertex(vertex, G);* |
| *void V::examine_edge(edge, G);* |
| *void V::tree_edge(edge, G);* |
| *void V::non_tree_edge(edge, G);* |
| *void V::gray_target(edge, G);* |
| *void V::black_target(edge, G);* |

Table 5: Documentation for the BFS Visitor concept.

In the code below, an example use of the ***breadth_first_search*** function is presented. The vertices of a graph are output in breadth-first order by creating the ***test_vis*** visitor that overrides the function ***discover_vertex***; empty implementations of the other visitor functions are provided by ***default_bfs_visitor***. A graph is constructed using the ***AdjacencyList*** class, and then the call to ***breadth_first_search*** is made. The call site is the point where type checking occurs for the body of the ***breadth_first_search*** function template; function templates are not separately type checked. This type check ensures that the argument types satisfy the needs of the body of the generic function, but it does not verify that the types model the concepts required by the algorithm (because the needs of the body may be less than the declared constraints for the function).

```
typedef graph_traits<AdjacencyList>::vertex vertex;

struct test_vis : public default_bfs_visitor {
  void discover_vertex(vertex v, const AdjacencyList& g)
    { cout << v << " "; }
};

int main(int, char*[]) {
  int n = 7;
  typedef pair<int,int> E;
  E edges[] = { E(0,1), E(1,2), E(1,3), E(3,4),
                E(0,4), E(4,5), E(3,6) };
  AdjacencyList g(n, edges);
  vertex s = get_vertex(0, g);
  vector_property_map color(n, white);
  breadth_first_search(g, s, color, test_vis());
  return EXIT_SUCCESS;
}
```

## 4.2 Evaluation of C++ Generics

C++ templates succeed in enabling the expression of generic algorithms, even for large and complex generic libraries. It is relatively easy to convert concrete functions to function templates, and function templates are just as convenient for the client to call as normal functions. The traits mechanism provides a way to access associated types, an area where several other languages fail.

The C++ template mechanism, however, has some drawbacks in the area of modularity. The complete implementations of templates reside in header files (or an equivalent). Thus, users must recompile when template implementations change. In addition, at call sites to function templates, the arguments are not type checked against the interface of the function—the interface is not expressed in the code— but instead the body of the function template is type checked. As a result, when a function template is misused, the resulting error messages point to lines within the function template. The internals of the library are thus needlessly exposed to the user and the real reason for the error becomes harder to find.

Another problem with modularity is introduced by the C++ overload resolution rules. During overload resolution, functions within namespaces that contain the definitions of the types of the arguments are considered in the overload set ("argument-dependent lookup"). As a result, any function call inside a function template may resolve to functions in other namespaces. Sometimes this may be the desired result, but other times not. Typically, the operations required by the constraints of the function template are meant to bind to functions in the client's namespace, whereas other calls are meant to bind to functions in the namespace of the generic library. With argument-dependent lookup, these other calls can be accidentally hijacked by functions with the same name in the client's namespace.

Nevertheless, C++ templates still provide type safety with genericity; there is no need to use downcasts or similar mechanisms when constructing generic libraries. Of course, C++ itself is not fully type safe because of various loopholes that exist in the type system. These loopholes, however, are orthogonal to templates. The template system does not introduce new issues with respect to type safety.

Finally, since templates are purely a means for obtaining static (compile-time) polymorphism, there is no run-time performance penalty due to templates *per se*. Generic libraries, however, make heavy use of procedural and data abstraction which can induce run-time overheads, though good optimizing compilers are adept at at flattening these layers of abstraction. C++ can therefore be an excellent tool for applications where run-time efficiency is critical [44,47]. Heavy use of templates can sometimes lead to significant increases in executable size, although there are programming idioms that ameliorate this problem.

## 5. GRAPH LIBRARY IN ML

Generic programs in Standard ML leverage three language features: structures, signatures, and functors. Structures group program components into named modules. They manage the visibility of identifiers and at the same time package related functions, types, values, and other structures. Signatures constrain the contents of structures. A signature prescribes what type names, values, and nested structures must appear in a structure. A signature also prescribes a type for each value, and a signature for each nested structure. In essence, signatures play the same role for structures as types play for values. Functors are templates for creating new structures and are parameterized on values, types, and structures. Multiple structures of similar form can be represented using a single functor that emphasizes characteristics the structures hold in common. Differences between these structures are captured by the functor's parameters. Functors represent ML's primary mechanism for generics. As illustrated in the following, structures, signatures, and functors together enable generic programming.

## 5.1 Implementation

Concepts are expressed in ML using signatures. The following code shows ML representations of graph concepts for the breadth-first search algorithm:

```
signature GraphSig =
sig
    type graph_t
    eqtype vertex_t
end

signature IncidenceGraphSig =
sig
    include GraphSig
    type edge_t
    val out_edges : graph_t → vertex_t → edge_t list
    val source : graph_t → edge_t → vertex_t
    val target : graph_t → edge_t → vertex_t
end
```

```
signature VertexListGraphSig =
sig
    include GraphSig
    val vertices : graph_t → vertex_t list
    val num_vertices : graph_t → int
end
```

For signature names, we use the convention of affixing ***Sig*** to the end of corresponding concept names. The ***GraphSig*** signature represents the Graph concept and requires ***graph_t*** and ***vertex_t*** types. It also requires ***vertex_t*** to be an ***equality type***, meaning ***vertex_t*** values can be compared using the = operator.

***IncidenceGraphSig*** and ***VertexListGraphSig*** demonstrate concept refinement in ML. The clause ***include GraphSig*** in each signature imports the contents of the ***GraphSig*** signature. The ***include*** directive cannot, however, represent all refinements between concepts. Though a signature may include more than one other signature, all included signatures must declare different identifiers. Consider the following code:

```
(* ERROR: VertexListGraphSig and IncidenceGraphSig overlap *)
signature VertexListAndIncidenceGraphSig =
sig
    include VertexListGraphSig
    include IncidenceGraphSig
end
```

This example shows an incorrect attempt to describe a Vertex List And Incidence Graph concept that refines both the Vertex List Graph and Incidence Graph concepts. The ML type system rejects this example because both ***VertexListGraphSig*** and ***IncidenceGraphSig*** share the ***vertex_t*** and ***graph_t*** names from the ***GraphSig*** signature. To work around this issue, an algorithm that would otherwise require a model of the Vertex List and Incidence Graph concept instead requires two arguments, a model of Vertex List Graph and a model of Incidence Graph, and places additional restrictions on those arguments. The implementation of breadth-first search in ML, shown later, demonstrates this technique.

Program components that model concepts are implemented as structures. The following code shows the adjacency list graph implemented in ML:

```
structure ALGraph =
struct
  datatype graph_t = Data of int * int list Array.array
  type vertex_t = int
  type edge_t = int * int

  fun create(nv : int) = Data(nv,Array.array(nv,[]))

  fun add_edge (G as Data(n,g),(src:int),(tgt:int)) =
      ( Array.update(g,src,tgt::Array.sub(g,src)); G )

  fun vertices (Data(n,g)) = List.tabulate(n,fn a => a);
  fun num_vertices (Data(n,g)) = n
  fun out_edges (Data(n,g)) v = map (fn n => (v,n)) (Array.sub(g,v))
  fun adjacent_vertices (Data(n,g),v) = Array.sub(g,v)
  fun source (Data(n,g)) (src,tgt) = src
  fun target (Data(n,g)) (src,tgt) = tgt

  fun edges (Data(n,g)) =
      #2(Array.foldl (fn (tgts:int list,(src,sofar:(int*int) list)) =>
                        (src+1,(map (fn n => (src,n)) tgts) @ sofar))
         (0,[]) g)
end;
```

The ***ALGraph*** structure encapsulates types that represent graph values and functions that operate on them. Because it meets the requirements of the ***GraphSig***, ***VertexListGraphSig***, and ***IncidenceGraphSig*** signatures, ***ALGraph*** is said to model the Graph, Vertex List Graph, and Incidence Graph concepts. ***ALGraph*** defines additional functions that fall outside the requirements of the three signatures. The ***create*** function, for example, constructs a value of type ***graph_t***, which represents a graph with ***nv*** vertices.

In ML, algorithms are implemented using functors. The following code illustrates the general structure of a generic breadth-first search implementation:

```
functor MakeBFS(Params : BFSPSig) =
struct
  fun breadth_first_search g v vis map = ...
end;
```

Generic algorithms are instantiated by way of functor application. When a functor is applied to parameters that satisfy certain requirements, it creates a new structure specialized for the functor parameters. The ***MakeBFS*** functor takes one parameter, a structure that fulfills the requirements of the following signature:

```
signature BFSPSig =
sig
  structure G1 : IncidenceGraphSig
  structure G2 : VertexListGraphSig
  structure C : ColorMapSig
  structure Vis : BFSVisitorSig
  sharing G1 = G2 = Vis
  sharing type C.key_t = G1.vertex_t
end
```

The signature dictates that ***Params*** must contain four nested structures, each corresponding to an algorithm parameter. ***BFSPSig*** enforces concept requirements by constraining its nested structures with signatures. The ***G1*** structure, for example, is constrained by the ***IncidenceGraphSig*** signature.

The breadth-first search algorithm ideally requires a graph type argument that models both the Incidence Graph and Vertex List Graph concepts. Because the signatures that represent these two concepts cannot be composed, the implementation requires two arguments, constrained by the signatures ***IncidenceGraphSig*** and ***VertexListGraphSig*** respectively. When the ***MakeBFS*** functor is applied, the same structure is bound to both type parameters.

In addition to listing required structures, ***BFSPSig*** specifies that some type names in the structures must refer to identical types. These are denoted as ***sharings***. Two sharings appear in the ***BFSPSig*** signature. The first is a ***structure sharing*** among ***G1***, ***G2***, and ***Vis***. It states that if the three structures share any nested element name in common, then the name must refer to the same entity for all three structures. For example, each of the three structures is required by its signature to contain a nested type ***vertex_t***. The sharing requires that ***G1.vertex_t***, ***G2.vertex_t***, and ***Vis.vertex_t*** must refer to the same type. The second sharing, a ***type sharing***, declares that ***C.key_t*** and ***G1.vertex_t*** must be the same type. Sharings emphasize that in addition to the signature requirements placed on each substructure of ***Params***, certain relationships between structures must also hold.

ML supports multi-parameter functors, but it does not support sharing specifications among the parameters. As a workaround, functors that implement generic algorithms accept a single structure parameter whose signature lists the algorithm's arguments and specifies the necessary relationships among them. Since the structure argument to the functor can be defined at the point of application, the single parameter solution is reasonable.

The following code shows a call to ***breadth_first_search***:

```
structure BFS =
MakeBFS(struct
            structure G1 = ALGraph
            structure G2 = ALGraph
```

```
        structure C = ALGColorMap
        structure Vis = VisitImpl
    end)

BFS.breadth_first_search g src (VisitImpl.create())
                               (ALGColorMap.create(graph));
```

First, the algorithm is instantiated by applying ***MakeBFS*** to a structure, defined in place, that meets ***BFSPSig***'s requirements. The ***ALGraph*** structure is used to match both the ***IncidenceGraphSig*** and ***VertexListGraphSig*** signatures. Although this is awkward, it avoids the explicit declaration of a ***VertexListAndIncidenceGraphSig*** signature, which cannot be constructed by composing the two mentioned signatures. The ***ALGColorMap*** structure models the Read/Write Map concept. The ***VisitImpl*** structure models the BFS Visitor concept and encapsulates user-defined callbacks. The three structures together meet the sharing requirements of ***BFSPSig***. Application of the ***MakeBFS*** functor defines the ***BFS*** structure, which encapsulates a ***breadth_first_search*** function specialized with the above structures. Finally, ***BFS.breadth_first_search*** is called with parameters that match the now concrete type requirements.

## 5.2 Evaluation of ML

ML language mechanisms provide good support for generic programming. Signatures and structures conveniently express concepts and concept models using nested types and functions to implement associated types and valid expressions. The structure representation of concept models enables modularity by managing identifier visibility. Functors can express any generic algorithm of similar complexity to the described graph library algorithms. Signatures effectively constrain generic algorithms with respect to the concepts upon which the algorithms are parameterized. Sharing specifications enable separate type checking of generic algorithms and their call sites. They capture additional requirements on the concept parameters to an algorithm. All necessary sharing relationships between functor parameters must be declared explicitly. If not, ML will issue type checking errors when the functor is analyzed. When a functor is applied, ML verifies that its arguments also meet the sharing and signature requirements imposed on the functor.

Technically, functors are not the only means for implementing generic algorithms. ML programmers often use polymorphic functions and parameterized data types to achieve genericity. An example of this style of programming follows.

```
(∗ concept ∗)
datatype 'a Comparable = Cmp of ('a → 'a → bool);

(∗ models ∗)
datatype Apples = Apple of int;
fun better_apple (Apple x) (Apple y) = x > y;

datatype Oranges = Orange of int;
fun better_orange (Orange x) (Orange y) = x > y;

(∗ algorithm ∗)
fun pick ((Cmp better):'a Comparable) (x:'a) (y:'a) =
    if (better x y) then x else y;

(∗ examples ∗)
pick (Cmp better_apple) (Apple 4) (Apple 3);
pick (Cmp better_orange) (Orange 3) (Orange 4);
```

This example implements the ***better*** algorithm in terms of the Comparable concept. Here a concept is realized using a parameterized data type that holds a table of functions or ***dictionary***. The concept's associated types are the data type's parameters, and its valid expressions are the dictionary functions. In addition to other values, a generic algorithm takes a dictionary for each concept model it requires. The algorithm is then implemented in terms of the functions from the dictionaries.

This style of generic programming in ML, though possible, is not ideal. In larger ML programs, managing dictionaries manually becomes cumbersome and increases the code base significantly. This situation is analogous to implementing virtual tables in C rather than leveraging the object-oriented programming features of C++. In fact, some Haskell environments lower programs that use generics (type classes) to equivalent Haskell programs that use this dictionary-passing style. Automating the mechanisms of generic programming is preferable to implementing them manually.

Using ML functors to implement generic algorithms enables the convenient application of algorithms to a variety of user-defined components. Functors in ML only require their arguments to conform structurally to the specified signatures. Since ML structures can implicitly conform to signatures, a structure need not be designed with a signature in mind. Thus, a generic ML algorithm, written in terms of signatures, can operate on any structures that meets its requirements.

In order to promote modularity, a language may allow program components that model concepts to be statically checked against concepts prior to their use with generic algorithms. When a structure is defined in ML, it may be constrained by a signature. In this manner a structure's conformity to a signature can be confirmed apart from its use in a generic algorithm. Constraining a structure with a signature limits its interface to that described by the signature. This may not be the desired result if the structure defines members that the signature does not declare. For example, if the ***ALGraph*** structure were declared:

```
structure ALGraph : IncidenceGraphSig = ...
```

then it would no longer meet the ***VertexListGraphSig*** requirements because ***vertices*** and ***num_vertices*** would not be visible.

Rather than constrain the structure directly, the conformity of ***ALGraph*** to the necessary signatures can be checked as shown in the following code outline:

```
structure ALGraph =
struct
  ...
end

structure ALGraphCheck1 : IncidenceGraphSig = ALGraph;
structure ALGraphCheck2 : VertexListGraphSig = ALGraph;
```

The structures ***ALGraphCheck1*** and ***ALGraphCheck2*** are both assigned ***ALGraph*** and constrained by the ***IncidenceGraphSig*** and ***VertexListGraphSig*** signatures respectively. Each of these structures confirms statically that ***ALGraph*** conforms to the corresponding signature without limiting access to its structure members. This technique as a side effect introduces the unused ***ALGraphCheck1*** and ***ALGraphCheck2*** structures.

As previously described, the ***include*** mechanism for signature combination in ML cannot express concept refinements that involve overlapping concepts. Ramsey [40] discusses this shortcoming and suggests language extensions to address it.

## 6. GRAPH LIBRARY IN HASKELL

The Haskell community uses the term "generic" to describe a form of generative programming with respect to algebraic datatypes [2, 15, 19]. Thus the typical use of the term "generic" with respect to Haskell is somewhat different from our use of the term. However, Haskell does provide support for generic programming as we have defined it here and that is what we present in this section.

The specification of the graph library in Figure 3 translates naturally into polymorphic functions in Haskell. In Haskell, a function is polymorphic if an otherwise undefined type name appears in the type of a function; such a type is treated as a parameter. Constraints on type parameters are given in the ***context*** of the function, i.e., the code between ***::*** and ⇒. The context contains ***class assertions***. In Haskell, concepts are represented with ***type classes***. Although the keyword Haskell uses is ***class***, type classes are not to be confused with object-oriented classes. In traditional object-oriented terminology, one talks of objects being instances of a class, whereas in Haskell, types are instances of type classes. A class assertion declares which concepts the type parameters must model. In Haskell, the term ***instance*** corresponds to our term ***model***. So instead of saying that a type models a concept, one would say a type is an instance of a type class.

## 6.1 Implementation

As with the previous languages, we focus on the interface of breadth-first search. The Haskell version of this function is shown below. The first line gives the name, and the following three are the context of the function. The function is curried; it has five parameters and the return type is ***a***, a user defined type for the output data accumulated during the search.

```
breadth_first_search ::
  (VertexListGraph g v, IncidenceGraph g e v,
   ReadWriteMap c v Color,
   BFSVisitor vis a g e v) ⇒
  g → v → c → vis → a → a
```

The following are the type classes for the Graph, Incidence Graph, and Vertex List Graph concepts:

```
class Graph g e v | g → e, g → v where
  src :: e → g → v
  tgt :: e → g → v

class Graph g e v ⇒ IncidenceGraph g e v where
  out_edges :: v → g → [e]
  out_degree :: v → g → Int

class VertexListGraph g v | g → v where
  vertices :: g → [v]
  num_vertices :: g → Int
```

The use of contexts within type class declarations is the Haskell mechanism for concept refinement. Here we have ***IncidenceGraph*** refining the ***Graph*** concept.

Associated types are handled in Haskell type classes differently from C++ or ML. In Haskell, all the associated types of a concept must be made parameters of the type class. Thus, the graph concepts are parameterized not only on the main graph type, but also on the vertex and edge types. If we had used an iterator abstraction instead of plain lists for the out-edges and vertices, the graph type classes would also be parameterized on iterator types. In Haskell 98, type classes are restricted to a single parameter, but most Haskell implementations support multiple parameters. The ***g*** → ***e*** denotes a functional dependency [20,22]. That is, for a given graph type ***g*** there is a unique edge type. Without functional dependencies it would be difficult to construct a legal type in Haskell for ***breadth_first_search***.

The ***BFSVisitor*** type class, shown below, is parameterized on the graph, queue, and output type ***a***. The queue and output type are needed because Haskell is a pure functional language and any state changes must be passed through explicitly, as is done here, or implicitly using monads. The visitor concept is also parameterized on the vertex and edge types because they are associated types of

```
data AdjacencyList = AdjList (Array Int [Int])
    deriving (Read, Show)
data Vertex = V Int deriving (Eq, Ord, Read, Show)
data Edge = E Int Int deriving (Eq, Ord, Read, Show)

adj_list :: Int → [(Int,Int)] → AdjacencyList
adj_list n elist =
  AdjList (accumArray (++) [] (0, n − 1)
                      [ (s, [t]) | (s,t) ← elist])

instance Graph AdjacencyList Edge Vertex where
  src (E s t) g = V s
  tgt (E s t) g = V t

instance IncidenceGraph AdjacencyList Edge Vertex where
  out_edges (V s) (AdjList adj) = [ E s t | t ← (adj!s) ]
  out_degree (V s) (AdjList adj) = length (adj!s)

instance VertexListGraph AdjacencyList Vertex where
  vertices (AdjList adj) = [V v | v ← (iota n) ]
    where (s,n) = bounds adj
  num_vertices (AdjList adj) = n+1
    where (s,n) = bounds adj
```

Figure 6: Simple adjacency list implementation.

the graph. The ***BFSVisitor*** type class has default implementations that do nothing.

```
class (Graph g e v) ⇒ BFSVisitor vis q a g e v where
  discover_vertex :: vis → v → g → q → a → (a,q)
  examine_edge :: vis → e → g → q → a → (a,q)
  ...
  discover_vertex vis v g q a = (a,q)
  examine_edge vis e g q a = (a,q)
  ...
```

The implementation of the ***AdjacencyList*** type is shown in Figure 6. The ***AdjacencyList*** type must be explicitly declared to be an instance of the Incidence Graph and Vertex List Graph concepts.

The following shows an example use of the ***breadth_first_search*** function to create a list of vertices in breadth-first order.

```
n = 7::Int
g = adj_list n [(0,1),(1,2),(1,3),(3,4),(0,4),(4,5),(3,6)]
s = vertex 0

data TestVis = Vis
instance BFSVisitor TestVis q [Int]
  AdjacencyList Edge Vertex where
  discover_vertex vis v g q a = ((idx v):a,q)

color = init_map (vertices g) White

res = breadth_first_search g s color Vis ([]::[Int])
```

Here, the ***idx*** function converts a vertex from an ***AdjacencyList*** to an integer. At the call site of a polymorphic function, the Haskell implementation checks that the context requirements of the function are satisfied by looking for instance declarations that match the types of the arguments. A compilation error occurs if a match cannot be found.

## 6.2 Evaluation of Haskell Generics

In general, we found Haskell to provide good support for generic programming. The Haskell type class mechanism, with the extensions for multiple parameters in type classes and functional dependencies, provides a flexible system for expressing complex generic libraries. The type classes and polymorphic functions provide succinct mechanisms for abstraction, and invoking a polymorphic function is almost as easy as invoking a normal function (the client may need to make instance declarations).

The modularity provided by type classes is excellent. Name lookup for function calls within a generic function is restricted to the namespace of the generic function plus the names introduced by the constraints. Generic functions and calls to generic functions are type checked separately, each with respect to the interface of the generic function. However, type errors (we used the Hugs November 2002 implementation) tend to be difficult to understand. We believe this is because the Haskell type system is based on type inferencing. When the deduced type of the body of a generic function does not match the type annotation for the function, the error message points to the type annotation. However, the more important piece of information is which expression within the function body caused the mismatch.

There are two issues with regards to convenience of expression when using Haskell. First, the presence of associated types in the parameter list of a type class is burdensome, especially when type classes are composed. For example, the ***BFSVisitor*** type class has six parameters. However, only three parameters are needed in principle and the other three are associated types. Two of the parameters are associated types of the graph (edge and vertex) and one is an associated type of the visitor (its output type).

The second convenience issue is that clients of a polymorphic function must declare their types to be instances of the type classes used as constraints on the polymorphic function. This adds textual overhead to calling a generic function compared to a normal function. On the other hand, instance declarations add a level of safety by forcing clients to think about whether their types model the required concepts at a semantic as well as a syntactic level.

## 7. GRAPH LIBRARY IN EIFFEL

Eiffel supports generics through type parameterization of classes. Each formal type parameter can be accompanied by a constraint, a type to which the actual type argument must conform (the term Eiffel uses for substitutability). Type parameters follow the class name within square brackets; the arrow → attaches a constraint to a type parameter. If omitted, the constraint defaults to the ***ANY*** class, the root of the Eiffel class hierarchy. The constraining type may refer to other type parameters. An example of such constraint is ***GRAPH_EDGE[V]***, shown in Figure 7.

### 7.1 Implementation

Concepts are represented as deferred classes (cf. abstract classes in C++). To model a given concept, a type must inherit from the class representing the concept. The requirements in a concept definition often need to refer to associated types to constrain them. Eiffel does not have a mechanism to attach types to classes; thus, associated types are expressed as type parameters of the classes that represent concepts. The ***V*** and ***E*** parameters in Figure 8 are examples of this. Due to the lack of parameterized methods, generic algorithms are represented as parameterized classes that contain one method named ***go***.

The bodies of generic algorithms frequently use associated types of the algorithm's parameter types. For example, in the breadth-first search algorithm, the vertex and edge types of the graph type must be accessible. Generic algorithms access associated types by including them as additional type parameters. For example, in Figure 7 the type parameters ***V*** and ***E*** represent the associated vertex and edge types and are used as type arguments in the constraint of the graph type ***G***. Analogously, the same type parameters appear in the definitions of the corresponding concepts.

The interface of the breadth-first search algorithm, shown in Figure 7, is representative of the generic algorithms in the Eiffel graph library implementation. The graph type ***G*** must model the Vertex List Graph and Incidence Graph concepts. The combined set of requirements of these two concepts is encompassed in the class ***VERTEX_LIST_AND_INCIDENCE_GRAPH[V, E]***, shown in Figure 8.

```
class BREADTH_FIRST_SEARCH[V, E→ GRAPH_EDGE[V],
            G→ VERTEX_LIST_AND_INCIDENCE_GRAPH[V, E]]
feature
   go(g: G; src: V; color: READWRITE_MAP[V, INTEGER];
       vis: BFS_VISITOR[G, V, E]) is ...
```

Figure 7: Interface of the breadth-first search algorithm in Eiffel.

```
deferred class VERTEX_LIST_GRAPH[V]
feature
   vertices: ITERATOR[V] is deferred end
   num_vertices: INTEGER is deferred end
end

deferred class INCIDENCE_GRAPH[V, E]
feature
   out_edges(v: V) : ITERATOR[E] is deferred end
   out_degree(v: V) : INTEGER is deferred end
end

deferred class VERTEX_LIST_AND_INCIDENCE_GRAPH[V, E]
inherit
   VERTEX_LIST_GRAPH[V]
   INCIDENCE_GRAPH[V, E]
end

class ADJACENCY_LIST
inherit
   VERTEX_LIST_AND_INCIDENCE_GRAPH
      [INTEGER, BASIC_EDGE[INTEGER]]
feature {NONE}
   data : ARRAYED_LIST[LINKED_LIST[INTEGER]]
   ...
```

Figure 8: Two graph concepts and a class that conforms to these concepts. ***V*** and ***E*** stand for the vertex and edge types, respectively.

The graph library implementation in Eiffel deviates from the design described in Figure 3. The Eiffel implementation uses fewer type parameters. For example, the classes that implement graph concepts have no type parameters for vertex and edge iterator types. In early attempts to rigorously follow the original design, calls to generic algorithms were overly verbose. The reasons for this, explicit instantiation and the inability to properly represent associated types, are discussed in Section 7.2.1. The disadvantage of fewer type parameters is that the exact types of all arguments or return values of generic algorithms cannot be expressed. For example, in Figure 8, the return type of the ***vertices*** is ***ITERATOR[V]***, not the exact vertex iterator type. Similarly, the static type of the ***vis*** parameter in the ***BREADTH_FIRST_SEARCH*** algorithm in Figure 7 is not the exact type of the visitor object. This loss of type accuracy has no performance implications in the Eiffel compilation model, where exact types are not exploited for static dispatching. However, inexact types can result in situations where either downcasting or relying on covariant changes in type parameters is needed. Covariant change in formal parameters of methods is a controversial feature of Eiffel that leads to type safety problems. Covariance in type parameters is not without problems either, as discussed in Section 7.2.2.

## 7.2 Evaluation of Eiffel Generics

Eiffel supports type checking generic classes independent of their use. This allows type errors to be caught before a generic class is instantiated. Furthermore, generic classes are compiled separately. Regardless of the number of instantiations, the compiled code contains one realization of each generic class.

### *7.2.1 Abstraction and Modularity*

Three factors affect Eiffel's support for abstraction and modularity: instantiation of generic algorithms, language mechanisms for representing and accessing associated types, and the lack of multiple generic constraints.

Eiffel requires explicit instantiation: the caller of a generic algorithm must pass both actual arguments and type arguments. Assuming the arguments have the following types:

```
g: ADJACENCY_LIST; src: INTEGER;
color: HASH_MAP[INTEGER, INTEGER]; vis: MY_BFS_VISITOR
```

the call to the breadth-first search algorithm in Eiffel is:

```
bfs: BREADTH_FIRST_SEARCH
        [INTEGER, BASIC_EDGE[INTEGER], ADJACENCY_LIST]
  ...
create bfs
bfs.go(g, src, color, vis)
```

Explicit instantiation is tedious when the number of type parameters is large. Furthermore, an explicitly instantiated call to a generic algorithm carries unnecessary dependencies on implementation details of the parameters. Section 10.4 covers both of these issues.

Associated types and constraints between them are part of concept requirements but Eiffel classes cannot properly encapsulate these requirements. Instead, every time a concept is used as a constraint, all of its associated types and their constraints must be stated explicitly. This is a common problem for Eiffel, Java generics, and Generic C#, and is discussed in greater length in Section 10.2. Eiffel *Anchored types* [30, chapter 12] do not provide a solution because they cannot express arbitrary dependencies between types.

Another set of problems arises as the combined effect of explicit instantiation and accessing associated types using type parameters. First, the problem of verbose calls to generic algorithms is exacerbated. When instantiating generic algorithms explicitly inside other generic components, the type arguments are often themselves instances of generic classes. As a result, specifying the type arguments explicitly can become a significant programming overhead. In an earlier implementation that followed the original library design the graph search algorithm had eight type parameters. In Dijkstra's shortest-path algorithm, counting the nested parameters, the internal call to the graph search algorithm required 35 type arguments. This effect made us alter the library design to reduce the number of type parameters, as described above in Section 7.1. Second, the associated types are dependent on the implementation details of the concrete types. Explicit instantiation spreads this dependency to call sites of generic algorithms. Sections 10.3 and 10.4 demonstrate this using examples and discuss the issue in more detail.

A type parameter constraint must be a single class. Classes that represent combinations of concepts are the obvious mechanism to work around the lack of multiple constraints. Thus, instead of the two classes ***VERTEX_LIST_GRAPH*** and ***INCIDENCE_GRAPH***, the graph type ***G*** in Figure 8 is required to derive from the class ***VERTEX_LIST_AND_INCIDENCE_GRAPH***. The requirements of generic algorithms determine which combinations of concepts require such classes. Adding a new algorithm that requires a previously unused combination of concepts necessitates the creation of a new class for this combination. The change propagates to all subclasses of this combination, including concrete graph types. Multiple generic constraints is an open issue in Eiffel standardization as a potential future addition to the language [32]. Using multiple constraints, the requirements for the ***G*** parameter could be written as:

```
G→ {VERTEX_LIST_GRAPH[V]; INCIDENCE_GRAPH[V, E]}
```

### *7.2.2 Static Type Safety*

Eiffel has been criticized for type-safety problems [8, 12]. In particular, type conformance checking is based on the subclass relation, which allows instance variables and parameters of routines to change covariantly. Under these rules the subclass relation does not imply substitutability, and additional measures must be taken to guarantee type-safety. The suggested approaches include a link-time *system validity check* [30] that requires a whole-program analysis, and a ban of so-called *polymorphic catcalls* [31]. To our knowledge, no publicly available compiler implements either of these analyses. Whether the polymorphic catcall check would be too conservative and rule out useful programs is not clear. A recent proposal [16] suggests requiring programmers to add explicit handler functions for each potentially type-unsafe program point. Work on applying this scheme to existing libraries and evaluating whether it provides a satisfactory solution is under way [16].

Eiffel applies covariance to generic parameters as well, which makes the type-safety problems a concern. According to the Eiffel conformance rules, type ***B[U]*** conforms to the type ***A[T]*** if the generic class ***B*** derives from ***A***, and if ***U*** conforms to ***T***. This either leads to type safety problems, or if they are avoided with overly conservative assumptions, to the rejection of useful and reasonable code. It is not too difficult to fabricate an example of the former case, leading to a program crash, but in practice we ran into the latter issue repeatedly. Here is an example that demonstrates the problem:

```
deferred class READWRITE_MAP[KEY, VALUE]
    ...
  put (k: KEY; val: VALUE) is deferred end
    ...
end

class WHITEN_VERTEX
        [V, CM → READWRITE_MAP[V, INTEGER]]
  inherit COLORS end
feature
  go(v: V; color_map : CM) is
  do
    color_map.put(v, WHITE);
  end
end
```

Several graph algorithms attach state information to vertices using property maps. The state is represented as integer constants ***WHITE***, ***BLACK***, and ***GRAY***. The example shows a generic algorithm for setting the state of a given vertex to ***WHITE***. The ***WHITEN_VERTEX*** algorithm takes two method parameters, the vertex ***v*** and the property map ***color_map***, and their types as type parameters ***V*** and ***CM***. The call ***color_map.put(v, WHITE)*** in the example fails. The map ***color_map*** is of some type ***CM*** that inherits from ***READWRITE_MAP[V, INTEGER]*** and thus one may expect the signature of the ***put*** method to be ***put(key: V; val: INTEGER)***. However, due to possible covariance of type parameters, the compiler must assume differently. Suppose the generic class ***MY_MAP[A, B]*** inherits from ***READWRITE_MAP[A, B]***, ***YOUR_VERTEX*** inherits from ***MY_VERTEX***, and ***YOUR_INT*** from ***INTEGER***. Now ***MY_MAP[YOUR_VERTEX, YOUR_INT]*** conforms to the class ***READWRITE_MAP[MY_VERTEX, INTEGER]***, making the following instantiation seemingly valid:

```
WHITEN_VERTEX[MY_VERTEX,
  MY_MAP[YOUR_VERTEX, YOUR_INTEGER]]
```

In this instantiation, the ***put*** method has the signature ***put(k: YOUR_VERTEX; val: YOUR_INTEGER)***, but is called with arguments of types ***MY_VERTEX*** and ***INTEGER***. This is an obvious error; to prevent errors such as this, some compilers, as an attempt to partially solve the covariance problem, reject the definition of the ***go*** method in the ***WHITEN_VERTEX*** class, at the same time disallowing its legitimate uses. Other compilers accept the definition, making programs vulnerable to uncaught type errors. In sum, the covariance rule causes code that seems to be perfectly reasonable to be rejected, or leads to type unsafety.

An immediate fix to this situation is to change the type of the ***color_map*** parameter:

```
go(color_map: READWRITE_MAP[V, INTEGER];
   v: INTEGER)
```

Now ***color_map*** is of type ***READWRITE_MAP[V, INTEGER]***, and the ***put*** signature is guaranteed to be ***put(k: V; val: INTEGER)***. However, along with this change, the exact type of the actual argument bound to ***color_map*** is lost. In this particular case, that would not be critical. However, suppose the algorithm kept the original map intact and returned a modified copy of the map as a result. The signature of such a method would be:

```
go(color_map: READWRITE_MAP[V, INTEGER];
   v: INTEGER) : READWRITE_MAP[V, INTEGER]
```

Regardless of the type of the actual argument bound to ***color_map***, the return value is coerced to ***READWRITE_MAP[V, INTEGER]***, losing all capabilities the original type potentially had.

The interface of Johnson's algorithm illustrates how covariance can be exploited to decrease the number of type parameters. One of the parameters of Johnson's algorithm is a map of maps storing distances between vertex pairs in a graph. The type constraint for this argument is:

```
READ_MAP[V, READWRITE_MAP[V, DISTANCE]];
```

A concrete type, such as

```
MY_MAP[INTEGER, MY_MAP[INTEGER, REAL]]
```

where ***V*** is bound to ***INTEGER*** and ***DISTANCE*** to ***REAL***, does not conform to the above constraint without covariance.

Generic programming does not seem to fundamentally benefit from covariance on type parameters as a language feature. We resorted to covariance only to reduce difficulties arising from the lack of implicit instantiation and support for associated types. Particularly, we were able to reduce the number of type parameters in a few situations where it would not have been possible without covariance. More notably, however, the covariance rules reduced flexibility. To guarantee type safety, new restrictions must be introduced. These restrictions lead to the compiler rejecting code for reasons that are not easy for a programmer to discern.

## 8. GRAPH LIBRARY IN JAVA GENERICS

Java generics extend the Java programming language with type parameters for classes, interfaces, and methods [6]. They have been proposed as an official addition to Java, and are planned for version 1.5 of that language. For our implementation, we used a prerelease version (1.3) of the proposed generics for Java. With respect to generics, the differences between this version and the proposed generics for Java 1.5 would not a affect our evaluation. Java generics support parameterized classes and methods. Generic methods are implicitly instantiated. Type parameters can be constrained using subtyping, including multiple constraints on one parameter.

```
public interface VertexListGraph<Vertex,
    VertexIterator extends Iterator<Vertex>> {
  VertexIterator vertices();
  int num_vertices();
}

public interface IncidenceGraph<Vertex, Edge,
  OutEdgeIterator extends Iterator<Edge>> {
  OutEdgeIterator out_edges(Vertex v);
  int out_degree(Vertex v);
}

public interface VertexListAndIncidenceGraph<...>
  extends VertexListGraph<...>, IncidenceGraph<...> {}

public class adjacency_list
  implements VertexListAndIncidenceAndEdgeListGraph<
    Integer, simple_edge<Integer>, Iterator<Integer>,
    Integer, Iterator<simple_edge<Integer>>, Integer,
    Iterator<simple_edge<Integer>>> {
  ...
}
```

Figure 9: Three interfaces representing graph concepts in Java, and the adjacency list data structure that models these concepts.

### 8.1 Implementation

In Java generics, we represent concepts with interfaces. A type declares that it models a concept by inheriting from the corresponding interface. Figure 9 shows the Java representations of three graph concepts, and an adjacency list graph data structure modeling these concepts. The ***implements*** clause makes ***adjacency_list*** a model of the Vertex List Graph and Incidence Graph concepts.

Generic algorithms are most straightforwardly expressed as functions. In languages without functions, they can be emulated by static methods. There are two choices for how to parameterize the method: either parameterized methods in non-parameterized classes or non-parameterized methods in parameterized classes. The first alternative has the advantage of implicit instantiation, while the second alternative requires the more verbose explicit specification of type arguments. Following the convention used within the Eiffel implementation in Section 7, the class name refers to the name of the algorithm, and the implementation method is named ***go***.

Java does not support type aliases, such as the ***typedef*** statement in C++, so associated types present a challenge. As in the Eiffel implementation, associated types are included as type parameters to the interface representing the concept. For example, the ***IncidenceGraph*** interface in Figure 9 has three associated types. The adjacency list graph type demonstrates connecting concrete associated types to the type parameters of the ***IncidenceGraph*** interface: the edge type is ***adj_list_edge<Integer>***, the vertex type is ***Integer***, etc.

Figure 10 shows the interface for the breadth-first search algorithm. This algorithm takes six type parameters and four method parameters. Three of the type parameters are types of method pa-

```
public class breadth_first_search {
  public static <
    Vertex,
    Edge extends GraphEdge<Vertex>,
    VertexIterator extends Iterator<Vertex>,
    OutEdgeIterator extends Iterator<Edge>,
    Visitor extends BFSVisitor,
    ColorMap extends ReadWriteMap<Vertex, Integer>>
  void go(VertexListAndIncidenceGraph<Vertex,Edge,
    VertexIterator,VerticesSizeType,OutEdgeIterator,
    DegreeSizeType> g, Vertex s, ColorMap c, Visitor vis);
}
```

Figure 10: Breadth-first search interface using Java generics

rameters; the other three represent associated types of the method parameter types and the graph type. A straightforward implementation would also have a type parameter for the graph type, but that approach is not feasible due to limitations in the type system for Java generics. These problems will be explained in detail later. Type parameters can be constrained to extend a class and optionally implement a set of interfaces.

Because of implicit instantiation, calling a generic method is no different from calling a non-generic method. For example, the breadth-first search algorithm is invoked by:

```
breadth_first_search.go(g, src, color_map, visitor);
```

## 8.2 Evaluation of Java Generics

Interfaces provide a mechanism to represent concepts within the language. Type parameter constraints can be expressed directly and are enforced by the compiler. Both arguments to generic algorithms and the bodies of those algorithms are checked separately against the concept requirements, leading to good separation between types used in generic algorithms and the implementations of those algorithms. Overall, Java generics provide enough support for generic programming to allow a type-safe implementation of the generic graph library. However, due to two problems related to subtyping-constrained polymorphism, and several inconveniences, generic programming in Java is both restricted and cumbersome.

The first problem originates from using inheritance to express the *models* relation. The inheritance relation is fixed when a class is defined. Consequently, existing classes cannot be made to model new concepts retroactively, unless their definitions can be modified. Section 10.1 explains how this problem significantly decreases modularity. The second problem is related to representing associated types. Java classes and interfaces can only encapsulate methods and member variables, not types. Hence, each concept's associated types are represented as separate type parameters. Referring to a concept in a generic algorithm requires repeating all its type parameters, including those not used in the algorithm. This results in unnecessarily lengthy code and repetition of the type constraints. Sections 10.2 and 10.3 discuss this problem in detail.

Only classes and interfaces can be used as type arguments; built-in types such as ***int*** and ***double*** cannot be used. One must instead use the bulkier ***Integer*** and ***Double*** classes. This results in both wordy and inefficient code. For example, ***int*** is a common choice for representing vertices, ***double*** for edge weights. Using the wrapper classes, the syntax for setting the weight of a graph edge is:

```
weight_map.set(new adj_list_edge(new Integer(3), new Integer(5)),
               new Double(3.4))
```

If built-in types were allowed as type arguments, the code would be simpler:

```
weight_map.set(new adj_list_edge(3, 5), 3.4)
```

Autoboxing is a planned feature for Java 1.5 which solves some of these problems [5]. This feature allows automatic conversion from a primitive type to its wrapper type, and a cast for the reverse conversion. This allows the shorter syntax above to be used, rather than requiring explicit creation of ***Integer*** and ***Double*** objects.

As another example of unnecessarily verbose code, Java syntax for declaring and initializing a variable requires the type of the variable to be written twice. For example, the code to create an integer object is:

```
Integer x = new Integer(3);
```

In this example, the name of the type is short. However, this is often not the case in generic programming libraries. Many types used in the graph library implementation span several lines; see Section 10.5 for an example. Repetition of such names is not only tedious, but increases the possibility of errors. The repetition is unavoidable, as Java does not have a type aliasing facility, such as ***typedef*** in C++.

Generic components and their uses can be compiled separately. A generic algorithm's implementation is type-checked once, rather than for each instantiation. Uses of a generic algorithm are checked to ensure that the concept requirements are satisfied, independently from the implementation of the generic algorithm. This allows for faster compilation, as well as for catching type errors as early as possible, and is a major advantage relative to C++.

Java generics provide a type-safe generic programming framework. For backwards compatibility, the language is implemented using *type erasure*, which forces some type checks to be performed at run-time, which can have an effect on performance. Moreover, certain type constraints cannot be expressed with a type-erasure-based language. These restrictions are known but cannot be resolved in Java generics without losing backward compatibility to the current Java language definition [7]. None of the restrictions of type erasure were encountered in the graph library implementation. Alternative, more flexible, approaches for adding generics to Java have been suggested [10, 36].

One major problem with the current type system for Java generics is that type arguments cannot be inferred from the constraints of a generic method. Without proper support for associated types, this kind of type inference is necessary for associated types to be simulated. This behavior is explicitly stated in the type system specification for Java generics [38]. Because of this behavior, the following code fails to compile:

```
public class java_associated_types {
  public static <Value, Iter extends Iterator<Value>>
  boolean m(Iter iter) {
    return (iter.next() == (Value)null);
  }

  public static void main(String[] args) {
    ArrayList<Integer> al = null;
    m(al.iterator());
  }
}
```

In the call to ***m*** in ***main***, values must be derived for the type parameters ***Value*** and ***Iter***. ***Iter*** is found to be ***Iterator<Integer>***, which is the iterator type for the container ***al***. Because the implicit instantiation mechanism does not examine type constraints, no value can be found for ***Value***; thus, it is given the default value ***Object***. Therefore the call to ***m*** fails because ***Iterator<Integer>*** is not a subtype of ***Iterator<Object>***. It would likely be possible to extend the Java type system to use type parameter constraints to

```
public class breadth_first_search {
  public static <
    G extends VertexListGraph<Vertex, VertexIterator>
            & IncidenceGraph<Vertex, Edge, OutEdgeIterator>,
    Vertex,
    Edge extends GraphEdge<Vertex>,
    VertexIterator extends Iterator<Vertex>,
    OutEdgeIterator extends Iterator<Edge>,
    Visitor extends BFSVisitor<G, Vertex, Edge>,
    ColorMap extends ReadWriteMap<Vertex, Integer>>
  void go(G g, Vertex s, ColorMap c, Visitor vis);
}
```

Figure 11: Breadth-first search interface using Java generics (when improved type system is used)

```
public interface VertexListGraph<Vertex,VertexIterator>
    where VertexIterator: IEnumerable<Vertex> {
  VertexIterator vertices();
  int num_vertices();
}
public interface IncidenceGraph<Vertex, Edge, OutEdgeIterator>
    where OutEdgeIterator: IEnumerable<Edge> {
  OutEdgeIterator out_edges(Vertex v);
  int out_degree(Vertex v);
}
public interface VertexListAndIncidenceGraph
    <Vertex, Edge, VertexIterator, OutEdgeIterator>
  : VertexListGraph<Vertex, VertexIterator>,
    IncidenceGraph<Vertex, Edge, OutEdgeIterator>
  where Edge: GraphEdge<Vertex>,
        VertexIterator: IEnumerable<Vertex>,
        OutEdgeIterator: IEnumerable<Edge> {}
public class adjacency_list
  : VertexListAndIncidenceGraph<int, adj_list_edge<int>,
    IEnumerable<int>, IEnumerable<adj_list_edge<int> > > {
  ...
}
```

Figure 12: Generic C# representations of three graph concepts and a type that models the concepts.

aid in implicit instantiation, which would allow the correct value of ***Integer*** to be found for ***Value***. With this modification, the interface to breadth-first search in Java could be written as shown in Figure 11. The current design of implicit instantiation in C# appears to follow similar rules, and is likely to have the same problem. A language with proper support for associated types would not need to be able to derive type parameter values from constraints at all, however.

In summary, the Java language, extended with generics, provides solid support for generic programming, but many problems make writing generic programs tedious and restricted.

## 9. GRAPH LIBRARY IN GENERIC C#

Generic C# extends the C# programming language with parameterized classes, interfaces, and methods [24]. Apart from minor syntactic differences, generic class and method definitions are similar to those in Java generics. Their use for generic programming is analogous as well: concepts are represented using interfaces, and a class declares that it models a concept by implementing the interface representing that concept. Full separate type checking and compilation are supported. Generics are a planned feature for the Whidbey release of C# [14], and a prototype implementation, named Gyro, has already been released [34]. Our implementation of the graph library used version 20020823 of Gyro.

```
public class breadth_first_search {
  public static void go<G, Vertex, Edge, VertexIterator,
      OutEdgeIterator, ColorMap, Visitor>
    (G g, Vertex s, ColorMap c, Visitor vis)
      where G: VertexListAndIncidenceGraph<
                  Vertex, Edge, VertexIterator,
                  OutEdgeIterator>,
            Edge: GraphEdge<Vertex>,
            VertexIterator: IEnumerable<Vertex>,
            OutEdgeIterator: IEnumerable<Edge>,
            ColorMap: ReadWriteMap<Vertex, ColorValue>,
            Visitor: BFSVisitor<G, Vertex, Edge>;
}
```

Figure 13: Breadth-first search interface in Generic C#. The ***IEnumerable*** interface provides the iteration mechanism in C#.

```
breadth_first_search.go<
  adjacency_list, int, adj_list_edge<int>,
  IEnumerable<int>,
  IEnumerable<adj_list_edge<int> >,
  my_bfs_visitor<adjacency_list, int, adj_list_edge<int> > >
(graph, src_vertex, color_map, visitor);
```

Figure 14: Call to breadth-first search algorithm in Generic C#.

### 9.1 Implementation

Figure 12 shows the interfaces representing the Vertex List Graph and Incidence Graph concepts, and an interface representing the combined requirements of these two concepts. The combined interface ***VertexListAndIncidenceGraph*** is necessary because the Gyro prototype implementation of Generic C# does not support multiple constraints on one type parameter. Furthermore, a graph data structure modeling these concepts is shown.

The interface to the breadth-first search algorithm is shown in Figure 13. The example invocation of this algorithm in Figure 14 illustrates the most significant difference between the current prototypes of Java generics and Generic C#: the Gyro implementation of Generic C# does not support implicit instantiation.

### 9.2 Evaluation of Generic C#

Generic C# avoids some of the inconveniences of Java generics; particularly, primitive types can be used as arguments to generic components. A severe drawback of the Gyro implementation of Generic C# is the lack of implicit instantiation. The need to explicitly specify all type arguments to generic methods leads to unnecessary code duplication and a loss of modularity, to be discussed in Section 10.4. Consequently, the Gyro implementation of Generic C# allows similar levels of abstraction and type safety as Java generics, but with substantially less modularity. The proposed standard for C# generics includes implicit instantiation, but cannot infer type parameter values based on constraints. Thus, the same problems that occurred with Java generics are also likely with the proposed C# generics.

The Gyro implementation of Generic C# does not support more than one constraint on a type parameter. Thus, if a type parameter must model more than one concept, combination interfaces must be created. The ***VertexListAndIncidenceGraph*** interface is an example of such a combination interface. The problems with this strategy are discussed in the evaluation of Eiffel generics, in Section 7.2.1. Support for more than one constraining type is in the current specification of generics for C#, and is planned for the Whidbey release of .NET [14].

## 10. DISCUSSION

The following sections describe five specific problems, each of which was encountered with more than one of the surveyed languages. The first problem concerns the type constraint mechanism and evolving software systems, especially when new algorithms and concepts are created. The second problem regards limitations in the use of subtyping to constrain type parameters. Third, we discuss the repercussions of the way associated types are accessed in the surveyed languages. Fourth, explicit instantiation combined with insufficient support for representing associated types can make generic function calls dependent on the implementation details of their argument types. Fifth, the lack of a mechanism for type aliasing is a source of unnecessary verbosity, especially when generic components must be explicitly instantiated.

### 10.1 Establishing the Modeling Relation

Type arguments to a generic algorithm must model the concepts that constrain the algorithm's type parameters. The languages under study use several different techniques to establish modeling relationships. Java generics, Generic C#, and Eiffel use subtyping at the point of class definition. Haskell requires an explicit ***instance*** declaration, independent of data type definition. Modeling relationships in ML are implicitly checked for structural conformance at generic function call sites. Any structure that meets the signature's requirements satisfies the modeling relationship. C++ provides no language feature for establishing modeling relationships. Type arguments are required only to provide the functionality that is used within a function template's body.

The modeling mechanisms used for Java, C#, Eiffel, and Haskell rely on named conformance. An explicit declaration links a concrete type to the concepts it models. Haskell differs from the others in that named conformance occurs separate from data structure definition. Modeling in ML relies on structural conformance. The names of concepts are irrelevant; only the requirements established by a signature matter. The modeling mechanisms in ML and Haskell worked well for implementing the graph library. ML's structural conformance has a small advantage in the area of convenience: the user of a generic function does not have to declare that his types model the required concepts. Named conformance, on the other hand, avoids problems with ***accidental conformance***. The canonical example of accidental conformance [28] is a ***rectangle*** class with ***move*** and ***draw*** methods, and a ***cowboy*** class with ***move***, ***draw***, and ***shoot*** methods. When modeling is based on structural conformance, a ***cowboy*** could be used where a ***rectangle*** is expected, possibly resulting in troublesome runtime errors. However, in our experience, accidental conformance is not a significant source of programming errors.

In languages where modeling is established by named conformance at type definition time, types cannot *retroactively model* concepts. Once a type is defined, the set of concepts that it models is fixed. Without modification to the definition, modeling relationships cannot be altered. This causes problems when libraries with interesting interdependencies are composed.

Figure 15 shows in C# an example of the retroactive modeling problem. In this example, library ***A*** defines a graph concept Vertex List Graph, as well as a graph data structure ***adjacency_list*** that models that concept. After this, library ***B*** creates an algorithm which requires only a subset of the Vertex List Graph concept from library ***A***, and library ***B*** defines a concept Vertex Number Graph corresponding to this subset. The problem is that ***adjacency_list*** should be a model of the new concept from library ***B***, but this is not possible in languages, such as Java, C#, and Eiffel, in which modeling relationships are fixed when a type is defined. Languages such as Haskell solve this by allowing modeling and refinement relationships to be established after the related types and concepts are defined. Languages such as ML and C++ do not encounter this problem, as they use structural conformance (and not named) to constrain type parameters.

```
namespace A {
  public interface VertexListGraph<...> where ... {
    VertexIterator vertices();
    int num_vertices();
  }
  public interface EdgeListGraph<...> where ... {
    EdgeIterator edges();
  }
  public class adjacency_list
  : VertexListGraph<...>, EdgeListGraph<...>
    { ... }
}
```

```
namespace B {
  public interface VertexNumberGraph<...> where ... {
    int num_vertices();
  }
  class bellman_ford {
    public static void go<G, ...>(g, ...)
    where G : VertexNumberGraph<...>, A.EdgeListGraph<...> { ... }
  }
}
```

```
namespace C {
  A.adjacency_list g;
  // Problem: A.adjacency_list does not inherit from
  // VertexNumberGraph
  B.bellman_ford.go<A.adjacency_list, ...>(g, ...);
}
```

Figure 15: An example showing need for retroactive modeling.

The problem of retroactive subtyping can be addressed by providing a language mechanism external to the class definition that establishes a subtyping relation [3, 27]. This is analogous to how the Haskell ***instance*** declaration established a modeling relation. Aspect oriented programming systems [26], such as AspectJ, allow modification of types independently of their original definitions. For example, an existing class can be modified to implement a newly-created interface using *static crosscutting* [25]. The *External Polymorphism* design pattern is an attempt to address this integration problem without changes to existing object-oriented languages [11]. Also, related problems are encountered with component interfaces in component software [9].

### 10.2 Subtyping is not Modeling or Refinement

In Java generics and Generic C#, concepts are approximated by interfaces, and similarly in Eiffel by deferred classes. The modeling relation between a type and a concept is approximated by the subtype relation between a type and an interface. The refinement relation between two concepts is approximated by interface extension. Constraints based on interfaces and subtyping, however, cannot correctly express constraints on multiple types.

Concepts commonly group constraints concerning multiple types. For example, the Vertex List Graph concept places constraints on a graph type, the vertex type of the graph, and a vertex iterator type. In particular, the Vertex List Graph concept requires that the vertex iterator type model the Iterator concept.

To express constraints on multiple types, an interface can be parameterized. For example, in Java generics, one might try the following approach for the definition of the ***VertexListGraph*** interface. A constraint is placed on the ***VertexIterator*** parameter:

```
interface VertexListGraph<
  Vertex, VertexIterator extends Iterator<Vertex>> {
  VertexIterator vertices();
  int num_vertices();
}
```

The problem with this approach is that referring by name to ***VertexListGraph*** is not sufficient to include the associated type constraint in that concept (in the context of a generic algorithm). For example, in the following ***breadth_first_search*** function, the ***VertexListGraph*** constraint should express the requirement that the ***VertexIterator*** type must model ***Iterator***:

```
class breadth_first_search {
 public static <
  G extends VertexListGraph<Vertex, VertexIterator>
          & IncidenceGraph<Vertex, Edge, OutEdgeIterator>,
  Vertex, Edge, VertexIterator, OutEdgeIterator,
  Visitor extends BFSVisitor<G, Vertex, Edge>,
  ColorMap extends ReadWriteMap<Vertex, Integer>>
 void go(G g, Vertex s, ColorMap c, Visitor vis);
}
```

This code fails to type check; in the reference to ***VertexListGraph***, the type argument ***VertexIterator*** fails to implement the interface ***Iterator***. Figure 10 shows the corrected version, which adds the constraint ***VertexIterator extends Iterator<Vertex>***. This repetition of the constraint means that interfaces fail to provide a way to organize and group constraints on multiple types. Instead the constituent constraints of such concepts must be repeated for every generic algorithm. This reduces the benefits of grouping constraints into concepts and is extremely verbose.

There is a similar problem with respect to approximating concept refinement with interface extension. For example, the following Java generics declaration is not legal because the ***OutEdgeIter*** type must be declared to extend ***Iterator<Edge>*** before instantiating ***IncidenceGraph***:

```
interface BidirGraph<Vertex, Edge, OutEdgeIter, InEdgeIter>
    extends IncidenceGraph<Vertex, Edge, OutEdgeIter> {...}
```

Here is the corrected version:

```
interface BidirGraph<Vertex, Edge,
    OutEdgeIter extends Iterator<Edge>, InEdgeIter>
    extends IncidenceGraph<Vertex, OutEdgeIter> {...}
```

## 10.3 Access to Associated Types

Generic functions need a mechanism to access associated types. For example, in breadth-first search, the type of the second parameter (the source of the search) is the vertex type associated with the graph type. The C++ prototype for ***breadth_first_search*** in Figure 4 shows how a traits class is used to map from the graph type to its vertex type. In ML, such a mapping is accomplished by nesting types within structures. The other languages we evaluated do not provide a direct mechanism for expressing functions over types which are able to dispatch on their inputs. For instance, an iterator type ***my_iterator*** might have value type ***int***, and another iterator type ***your_iterator*** might have value type ***double***.

Associated types can, however, be accessed in a generic function if they are added to the type parameter list of the generic function. The relationship between these extra type parameters, and the types to which they are associated, is established by parameterizing the interfaces (Java generics, C#, and Eiffel) or type classes (Haskell) that are used in the constraints of the generic function. This approach results in a significant increase in the verbosity of constraints on generic components. For example, the Java generics version of ***breadth_first_search*** shown in Figure 10 includes six type parameters, three of which are associated types of the graph. In contrast the C++ version of ***breadth_first_search*** has only three type parameters. The verbosity introduced by this approach compounds the problem of repeated constraints discussed in the previous section.

## 10.4 Implicit Instantiation

In languages that do not support implicit instantiation and cannot properly encapsulate associated types, such as Eiffel and the Gyro prototype of Generic C#, explicit instantiation results in overly verbose generic algorithm invocations. The call to the breadth-first search algorithm in Generic C# in Figure 14 demonstrates the problem: the programmer must specify types that could be deduced from the argument types. Representing associated types as type parameters increases the verbosity: in addition to the types of the arguments passed into a generic algorithm, the programmer must also specify all associated types. In the breadth-first search algorithm, all but the first and last two type arguments are associated types of the first type argument (the graph type). The code in Figure 16 serves as a stark, but not exaggerated, example of the combined effect of explicit instantiation and of representing associated types with type parameters.

The need to specify associated types also creates unnecessary implementation dependencies. As explained above, four of the type arguments in the call to the breadth-first search algorithm in Figure 14 are associated types of the first type argument. These types represent internal implementation details of the graph type. At each call to the algorithm these types must be explicitly specified. Consequently, changes to any implementation details require changes to user code at the call sites of generic algorithms. Consider the call to the breadth-first search algorithm in Eiffel that was shown in Section 7.2.1:

```
g: ADJACENCY_LIST; src: INTEGER;
color: HASH_MAP[INTEGER, INTEGER]; vis: MY_BFS_VISITOR
bfs: BREADTH_FIRST_SEARCH
        [INTEGER, BASIC_EDGE[INTEGER], ADJACENCY_LIST]
  ...
create bfs
bfs.go(g, src, color, vis)
```

The ***ADJACENCY_LIST*** class in Figure 8 uses ***INTEGER*** as the vertex type and ***BASIC_EDGE[INTEGER]*** as the edge type. Even though the breadth-first search algorithm does not take an edge object as a parameter, the edge type must still be specified. Consequently, changing the implementation of ***ADJACENCY_LIST*** to use a different edge type, such as ***MY_EDGE[INTEGER]***, requires the type of ***bfs*** to be changed accordingly:

```
bfs: BREADTH_FIRST_SEARCH
        [INTEGER, MY_EDGE[INTEGER], ADJACENCY_LIST]
```

Each call site to any graph algorithm that uses edges, even internally, and takes an argument of type ***ADJACENCY_LIST*** must be updated to reflect the new edge type.

## 10.5 Type Aliases

Type aliasing is a mechanism to provide an alternative name for a type (cf. the ***typedef*** keyword in C++). The parameterization of components introduces long type names, especially when parameterized components are composed. For example, in Generic C# the type of the visitor object used in the Dijkstra algorithm is:

```
dijkstra_visitor<G,
  mutable_queue<Vertex,
    indirect_cmp<Vertex, Distance, DistanceMap,
        DistanceCompare> >,
    WeightMap, PredecessorMap, DistanceMap,
    DistanceCombine, DistanceCompare, Vertex, Edge, Distance>
```

As in this example, type arguments are often instantiations of other parameterized components. Such types can be overlong, resulting in cluttered and unreadable code. Further, the long type name may appear many times. This six-line-long type must appear three times within the implementation of Dijkstra's algorithm. With type aliasing, a short name could be given to this type and thus reduce clutter in the code. Also, repeating the same type increases the probability of errors: changes to one copy of a type must be consistently applied to other copies. In addition to avoiding repetition of long type names, type aliases are useful for abstracting the actual types without losing static type accuracy.

Figure 16 shows the allocation of a visitor object and the call to the graph search algorithm that appears inside the Dijkstra implementation. Without type aliases, what should be a few lines of code is instead 27 lines. There is no mechanism for type aliases in Java, C#, or Eiffel.

Type aliasing is a simple but crucial feature for managing long type expressions commonly encountered in generic programming.

```
dijkstra_visitor<G,
  mutable_queue<Vertex,
    indirect_cmp<Vertex, Distance, DistanceMap,
      DistanceCompare> >,
  WeightMap, PredecessorMap, DistanceMap,
  DistanceCombine, DistanceCompare, Vertex, Edge,
  Distance> bfs_vis = new dijkstra_visitor<G,
    mutable_queue<Vertex,
      indirect_cmp<Vertex, Distance, DistanceMap,
        DistanceCompare> >,
      WeightMap, PredecessorMap, DistanceMap,
      DistanceCombine, DistanceCompare, Vertex, Edge,
      Distance>();

graph_search.go<
  G, Vertex, Edge, VertexIterator, OutEdgeIterator,
  hash_map<Vertex, ColorValue>,
  mutable_queue<Vertex,
    indirect_cmp<Vertex, Distance, DistanceMap,
      DistanceCompare> >,
  dijkstra_visitor<G,
    mutable_queue<Vertex,
      indirect_cmp<Vertex, Distance, DistanceMap,
        DistanceCompare> >,
    WeightMap, PredecessorMap, DistanceMap,
    DistanceCombine, DistanceCompare, Vertex, Edge,
    Distance> >
(g, s, color, Q, bfs_vis);
```

Figure 16: Lack of type aliases leads to unnecessarily lengthy code.

## 11. CONCLUSION: BEYOND `List<T>`

Mainstream object-oriented programming languages have begun to include support for generics. Java and C# generics have been accepted and are expected in future releases of those languages. Eiffel has supported generics from its inception. Generics have primarily been added to these languages to support type-safe polymorphic containers. Object-oriented programming techniques remain the primary mechanism for building abstractions in these languages; generics fill a small and specific need. The other languages we studied (C++, ML, and Haskell) support a broader, more powerful version of generic programming.

### 11.1 The C++ Experience

The C++ template system demonstrates that a flexible generics facility requires a different design than one intended merely for polymorphic containers. Although the other object-oriented languages studied here use subtyping to constrain generics, C++ rejected this approach because it lacks expressiveness and flexibility [46, §15.4]. Subtype-based constraints are adequate for polymorphic containers (which typically place few constraints on their parameters) but cannot convey the more complex constraints needed for generic programming. C++ also provides implicit function template instantiation, which enables convenient use of generic algorithms. Eiffel does not support implicit instantiation of generics, but it is planned as an addition to C# and Java.

C++ support for generic programming has limitations. The language does not support explicit expression of concepts. Documentation generally describes concept constraints without enforcing them. Argument dependent lookup obscures which operations a generic function may call. Finally, function templates are not type checked independent of their use. In spite of these flaws, commercial generic programs and libraries are successfully written in this language.

### 11.2 The ML and Haskell Experience

With respect to generics, ML and Haskell provide an informative contrast to the object-oriented languages. Both ML and Haskell provide polymorphic functions and data types, which together are sufficient to implement polymorphic data structures. Both languages also provide more powerful mechanisms: type classes in Haskell and functors in ML. The discussions of Java generics, Generic C#, and Eiffel suggest that Haskell and ML avoid some difficulties with implementing generic libraries that the object-oriented languages share. ML structures and signatures support associated types, component modularity, and static type safety. Haskell improves on ML generic programming support with implicit instantiation of generic functions. Both languages have disadvantages. ML lacks functions with constrained genericity aside from functions nested within functors. Haskell, though quite expressive, requires an awkward mechanism to express associated types. In general, these two languages provide fine substrates upon which to build generic libraries.

### 11.3 An Appeal to Language Designers

At the time of this writing, language support for generics continues to evolve. Updates to the C++ standard are currently under way. Generics have been promised for the next Java and C# standards. Eiffel is currently undergoing ECMA standardization. These languages have the opportunity to address their shortcomings with respect to support for generic programming. By doing so, they can augment the benefits that generics currently offer. Conscious consideration of support for flexible generics and expressive constraints can improve the ability of these languages to express modular and reusable program components.

## Acknowledgments

This work was supported by NSF grants EIA-0131354 and ACI-0219884, and by a grant from the Lilly Endowment. The authors would like to thank Todd Veldhuizen and the anonymous referees for their careful reading of this paper and for their well-considered comments.

## 12. REFERENCES

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

[2] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[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] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[5] G. Bracha and J. Bloch. *JSR 201: Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import*, December 2002. http://www.jcp.org/en/jsr/detail?id=201.

[6] G. Bracha, N. Cohen, C. Kemper, S. Marx, et al. *JSR 14: Add Generic Types to the Java Programming Language*, April 2001. http://www.jcp.org/en/jsr/detail?id=014.

[7] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 1998.

[8] K. B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Technical report, Williams College, 1996.

[9] M. Büchi and W. Weck. Compound types for Java. In *Proceedings of OOPSLA'98*, pages 362–373, October 1998. http://www.abo.fi/˜mbuechi/publications/OOPSLA98.html.

[10] R. Cartwright and G. L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 201–215. ACM, 1998.

[11] C. Cleeland, D. C. Schmidt, and T. H. Harrison. External polymorphism — an object structural pattern for transparently extending C++ concrete. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design*, volume 3 of *Software Pattern Series*. Addison-Wesley, 1997.

[12] W. R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):304–311, 1989.

[13] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] A. Hejlsberg. The C# programming language. Invited talk at Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2002.

[15] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In *Summer School on Generic Programming*, LNCS. Springer-Verlag, 2002/2003.

[16] M. Howard, E. Bezault, B. Meyer, D. Colnet, E. Stapf, K. Arnout, and M. Keller. Type-safe covariance: competent compilers can catch all catcalls. Apr. 2003.

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

[18] M. Jazayeri, R. Loos, D. Musser, and A. Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, Apr. 1998.

[19] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text: 2nd Int. School on Advanced Functional Programming*, volume 1129, pages 68–114. Springer-Verlag, Berlin, 1996.

[20] M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, number 1782 in LNCS, pages 230–244. Springer-Verlag, March 2000.

[21] S. P. Jones, J. Hughes, et al. *Haskell 98: A Non-strict, Purely Functional Language*, February 1999. http://www.haskell.org/onlinereport/.

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

[23] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

[24] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Snowbird, Utah, June 2001.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming*, vol. 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1994.

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

[28] B. Magnusson. Code reuse considered harmful. *Journal of Object-Oriented Programming*, 4(3), Nov. 1991.

[29] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.

[30] B. Meyer. *Eiffel: the language*. Prentice Hall, New York, NY, first edition, 1992.

[31] B. Meyer. Static typing. In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications*, pages 20–29. ACM Press, 1995.

[32] B. Meyer. The start of an Eiffel standard. *Journal of Object Technology*, 1(2):95–99, July/August 2002. www.jot.fm.

[33] Microsoft Corporation. Generics in C#, September 2002. Part of the Gyro distribution of generics for .NET [34].

[34] Microsoft Corporation. Generics for C# and .NET CLR, May 2003. http://research.microsoft.com/projects/clrgen/.

[35] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (Revised)*. MIT Press, 1997.

[36] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *the 24th ACM Symposium on Principles of Programming Languages*. Laboratory for Computer Science, MIT, January 1997.

[37] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[38] M. Odersky. Inferred type instantiation without prototypes for GJ. lampwww.epfl.ch/~odersky/ftp/local-ti.ps, Jan. 2002.

[39] R. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[40] N. Ramsey. Toward a calculus of signatures. http://www.eecs.harvard.edu/~nr/pubs/sigcalc-abstract.html, October 2001.

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

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

[43] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. The generic graph component library. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applicat ions*, pages 399–414. ACM Press, 1999.

[44] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

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

[46] B. Stroustrup. *Design and Evolution of C++*. Addison-Wesley, 1994.

[47] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.