JFP **17** (2): 145–205, 2007. © 2006 Cambridge University Press
doi:10.1017/S0956796806006198 First published online 19 December 2006 Printed in the United Kingdom

An extended comparative study of language support for generic programming

RONALD GARCIA[1], JAAKKO JÄRVI[2],
ANDREW LUMSDAINE[1], JEREMY SIEK[3]
and JEREMIAH WILLCOCK[1]

[1]*Open Systems Lab, Indiana University, Bloomington, IN, USA*
[2]*Texas A&M University, Computer Science, College Station, TX, USA*
[3]*Rice University, Computer Science, Houston, TX, USA*
(*e-mail:* {garcia,lums,jewillco}@osl.iu.edu, jeremy.g.siek@rice.edu, jarvi@cs.tamu.edu)

Abstract

Many modern programming languages support basic generics, sufficient to implement type-safe polymorphic containers. Some languages have moved beyond this basic support, and in doing so have enabled a broader, more powerful form of generic programming. This paper reports on a comprehensive comparison of facilities for generic programming in eight programming languages: C++, Standard ML, Objective Caml, Haskell, Eiffel, Java, C# (with its proposed generics extension), and Cecil. By implementing a substantial example in each of these languages, we illustrate how the basic roles of generic programming can be represented in each language. We also identify eight language properties that support this broader view of generic programming: support for multi-type concepts, multiple constraints on type parameters, convenient associated type access, constraints on associated types, retroactive modeling, type aliases, separate compilation of algorithms and data structures, and implicit argument type deduction for generic algorithms. We find that these features are necessary to avoid awkward designs, poor maintainability, and painfully verbose code. As languages increasingly support generics, it is important that language designers understand the features necessary to enable the effective use of generics and that their absence can cause difficulties for programmers.

1 Introduction

Generics are an increasingly popular and important tool for software development and many modern programming languages provide basic support for them. For example, the use of type-safe polymorphic containers is routine programming practice today. Some languages have moved beyond elementary generics to supporting a broader, more powerful form of generic programming, enabling the development of highly reusable algorithms. Such extensive support for generics has proven valuable in practice. Generic programming has been a particularly effective means for constructing reusable libraries of software components, one example of which is the Standard Template Library (STL), now part of the C++ Standard Library (Stepanov & Lee, 1994; ISO 1998). As the generic programming paradigm gains momentum, the need for a clear and deep understanding of the language issues increases. In particular, it is important to understand what language features are required to support this broader notion of generic programming.

To aid in this process, we present results of a study comparing eight programming languages that support generics: Standard ML (Milner *et al*., 1997), Objective Caml (Leroy, 2000), C++ (ISO, 1998), Haskell (Peyton Jones *et al*., 1999), Eiffel (Meyer, 1992), Java (Gosling *et al*., 2005), C# (Kennedy & Syme, 2001; Microsoft Corporation, 2002), and Cecil (Litvinov, 1998). Seven of these languages currently support generics, and they have been implemented and proposed for the next revision of C#. These languages were selected because they are widely used or represent the state of the art in programming languages with generics. This paper is a revised and extended version of an earlier paper (Garcia *et al*., 2003), featuring updated analyses and the addition of two languages, Objective Caml and Cecil.

Our goals for this study were the following:

- To understand which language features are necessary to support generic programming;
- To understand the extent to which specific languages support generic programming;
- To provide guidance for development of language support for generics; and
- To 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 any one language is "better" than any other. This paper is also not a comparison of generic programming to any other programming paradigm, be it object-oriented, functional, or otherwise.

To conduct the study, we designed a model library by extracting a small but significant example of generic programming from an existing generic library (the Boost Graph Library (Siek *et al*., 2002)). We then implemented the model library in each target language. This library exercises a variety of generic programming techniques 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 the different roles of generic programming. In addition, we evaluated each implementation with respect to software quality issues such as modularity, safety, and economy of expression. The full implementations for each language are available (Graph Library URL, 2005).

This process yielded two main results, which are the focus of this paper. First, we have identified how generic programming can be expressed in each language. Generic programming requires the fulfillment of several roles (introduced in the next section). Our study shows what features of each language can fulfill those roles. Second, we have identified eight language properties that are important to generic programming. Table 1 lists these properties and shows each language's level of support for them; the language properties themselves are described in Table 2. We find these properties necessary for the development of high-quality generic libraries. Incomplete support for them leads to awkward designs, poor maintainability, and painfully verbose code. As languages increasingly support generics, it is important that language

Table 1. *The level of support for important properties for generic programming in the evaluated languages. A black circle indicates full support, a white circle indicates poor support, and a half-filled circle indicates partial support. The rating of "-" in the C++ column indicates that C++ does not explicitly support the feature, but one can still program as if the feature were supported due to the permissiveness of C++ templates*

	C++	SML	OCaml	Haskell	Eiffel	Java	C#	Cecil
Multi-type concepts	-	●	○	●[*]	○	○	○	◒
Multiple constraints	-	◒	◒	●	○[†]	●	●	●
Associated type access	●	●	◒	●[★]	◒	◒	◒	◒
Constraints on assoc. types	-	●	●	●	◒	◒	◒	●
Retroactive modeling	-	●	●	●	○	○	◒	●
Type aliases	●	●	●	●	○	○	○	○
Separate compilation	○	●	◒	●	●	●	●	◒
Implicit arg. deduction	●	○	●	●	○	●	◒	◒

[*] Using the multi-parameter type class extension to Haskell (Peyton Jones *et al.*, 1997).
[★] Using the functional dependencies extension to Haskell (Jones, 2000).
[†] Planned language additions.

Table 2. *Glossary of evaluation criteria*

Criterion	Definition
Multi-type concepts	Multiple types can be simultaneously constrained.
Multiple constraints	More than one constraint can be placed on a type parameter.
Associated type access	Types can be mapped to other types within the context of a generic function.
Constraints on associated types	Concepts may include constraints on associated types.
Retroactive modeling	The ability to add new modeling relationships after a type has been defined.
Type aliases	A mechanism for creating shorter names for types is provided.
Separate compilation	Generic functions can be type checked and compiled independent of calls to them.
Implicit argument deduction	The arguments for the type parameters of a generic function can be deduced and do not need to be explicitly provided by the programmer.

designers understand the features necessary to provide powerful generics and that their absence causes serious difficulties for programmers.

The rest of this 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 11 present the individual implementations of the graph library in the selected languages. By explaining how the graph library

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.

Fig. 1. Definition of generic programming, from Jazayeri *et al.* (1998).

was implemented, we illustrate what features of each language fulfill the roles of generic programming. These sections also evaluate the level of support each language provides for generic programming. Section 12 discusses in detail the most important issues we encountered during the course of this study. Section 13 concludes the paper.

2 Generic programming

Definitions of generic programming vary. Typically, generic programming involves type parameters for data types and functions. Although 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 an earlier paper (Jazayeri *et al.*, 1998) reproduced in Figure 1. The next section discusses the terminology and techniques that have emerged to support generic programming.

Terminology

The notion of abstraction is fundamental to realizing generic algorithms: 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 syntactic and semantic requirements on one or more (abstract data) types (Austern, 1998). 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 specifies requirements on just one type – sometimes a concept involves multiple types and establishes relationships between them. In this sense, concepts generalize the interfaces of abstract data types.

Table 3. *The roles of language features used for generic programming*

Language	Algorithm	Concept	Refinement	Modeling	Constraint
C++	template	docs	docs	docs	docs
ML	functor	signature	include	implicit	param sig
OCaml	polymorphic function	class interface	inherit	interface	class type
Haskell	polymorphic function	type class	subclass	instance	context
Eiffel	generic class	deferred class	inherit	inherit	conformance
Java	generic method	interface	extends	implements	extends
C#	generic method	interface	inherit	inherit	inherit
Cecil	parameterized method	abstract object	subtyping	subtyping	type constraint

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

Traditionally, a concept consists of ***associated types***, ***valid expressions***, semantic constraints, and complexity guarantees. The associated types of a concept are opaque types that are required by the concept (e.g., used in valid expressions) and must be defined by any model of the concept. Valid expressions specify the operations that must be implemented for the modeling type (or types). Type systems typically do not include semantic constraints and complexity guarantees. For the purpose of this study, we thus state that for a type to model a concept, it suffices that the associated types and valid expressions specified by the concept are defined.

The basic roles of generic programming – generic algorithms, concepts, refinement, modeling, and constraints – are realized in different ways in our target programming languages. The specific language features used to support generic programming are summarized in Table 3.

Example

A simple example illustrates generic programming and its issues. The following is an example of a generic algorithm, realized as a C++ function template:

```
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 just any type can be used with ***pick***. The Comparable concept represents types that may be used with ***pick***. Unfortunately, C++ does not support concepts directly so naming conventions and documentation styles have been established to represent them (Austern, 1998). In particular, the Comparable concept is documented as follows:

Comparable concept	
better(a, b)	convertible ***bool***

The required valid expressions are in the left column, and requirements on their return types in the right column. A type ***T*** is a model of Comparable if it satisfies the above requirements, where objects ***a*** and ***b*** are of type ***T***. If we wish to make ***int*** model Comparable, we simply define a ***better*** function for ***int***s:

```
bool better(int x, int y) { return x > y; }
```

In C++ it is customary to identify concepts by appropriately naming template parameters. Under that guideline, ***pick*** would normally be written as follows:

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

Now consider two data types, ***apple*** and ***orange***:

```
struct apple {
  apple(int r) : rating(r) {}
  int rating;
};
bool better(const apple& x, const apple& y) { return x.rating > y.rating; }

struct orange {
  orange(const string& s) : name(s) { }
  string name;
};
bool better(const orange& x, const orange& y) {
  return lexicographical_compare(y.name.begin(), y.name.end(),
                                 x.name.begin(), x.name.end());
}
```

The ***apple*** and ***orange*** types model the Comparable concept implicitly via the existence of the ***better*** function for those types. The following illustrates calling the generic algorithm ***pick*** with arguments of types ***int***, ***apple***, and ***orange***:

```
int main(int, char*[]) {
  int i = 0, j = 2;
  apple a1(3), a2(5);
  orange o1("Navel"), o2("Valencia");

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

  return 0;
}
```

```
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 = Apple of int
  fun create n = Apple n
  fun better ((Apple x), (Apple y)) = x > y
end

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

Fig. 2. ***Apples*** to ***Apples*** in Standard ML.

```
class type comparable = object ('a) method better : 'a → bool end

class apple init = object (self : 'a)
  val value_ : int = init
  method better (y : 'a) = self#value > y#value
  method value = value_;
end

let pick ((x : #comparable as 'a), (y : 'a)) : 'a =
  if x#better y then x else y

let a1 = (new apple 3);;
let a2 = (new apple 1);;
let a3 = pick (a1, a2);;
```

Fig. 2 (cont.). ***Apples*** to ***Apples*** in OCaml.

Figure 2 shows how this example is implemented in the other seven languages.

3 A generic graph library

To evaluate support for generic programming, we implemented a generic graph library 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 (Dijkstra, 1959; Prim, 1957). The design presented here descends from the generic graph library reported in Lee *et al.* (1999), which evolved into the Boost Graph Library (BGL) (Siek *et al.*, 2002).

```
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 = Apple Int

instance Comparable Apple where
  better (Apple x, Apple y) = x > y

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

Fig. 2 (cont.). ***Apples*** to ***Apples*** in Haskell.

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

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

class APPLE inherit COMPARABLE[APPLE] end
create make
feature
  make(r: INTEGER) is do rating := r end
  better (y: APPLE) : BOOLEAN is do Result := rating > y.rating end
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
```

Fig. 2 (cont.). ***Apples*** to ***Apples*** in Eiffel.

```
public interface Comparable<T> {
   boolean better(T x);
}
public class pick {
   public static <T extends Comparable<T>>
   T go(T x, T y) {
      if (x.better(y)) return x; else return y;
   }
}
public class Apple implements Comparable<Apple> {
   public Apple(int r) { rating = r; }
   public boolean better(Apple y)
      { return rating > y.rating; }
   private 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);
   }
}
```

Fig. 2 (cont.). ***Apples*** to ***Apples*** in Java.

```
public interface Comparable<T> {
   bool better(T x);
}
public static class pick {
   public static T go<T>(T x, T y) where T : Comparable<T> {
      if (x.better(y)) return x; else return y;
   }
}
public class Apple : Comparable<Apple> {
   public Apple(int r) {rating = r;}
   public bool better(Apple y)
      { return rating > y.rating; }
   private int rating;
}
public static class main {
   public static int Main(string[ ] args) {
      Apple a1 = new Apple(3), a2 = new Apple(5);
      Apple a3 = pick.go(a1,a2);
      return 0;
   }
}
```

Fig. 2 (cont.). ***Apples*** to ***Apples*** in C# generics.

```
abstract object my_comparable['T];
signature better(:my_comparable['T], :T): bool;

template object apple isa my_comparable[apple];
field rank(@:apple): int;
method better(x@:apple, y@:apple): bool {x.rank > y.rank}

method pick(x: 'T <= my_comparable[T], y: T): T {
  better(x, y).if({x}, {y})
}

method test_pick(): void {
  let a1 := object isa apple {rank := 3};
  let a2 := object isa apple {rank := 5};
  let a3 := pick(a1, a2);
}
```

Fig. 2 (cont.). ***Apples*** to ***Apples*** in Cecil.

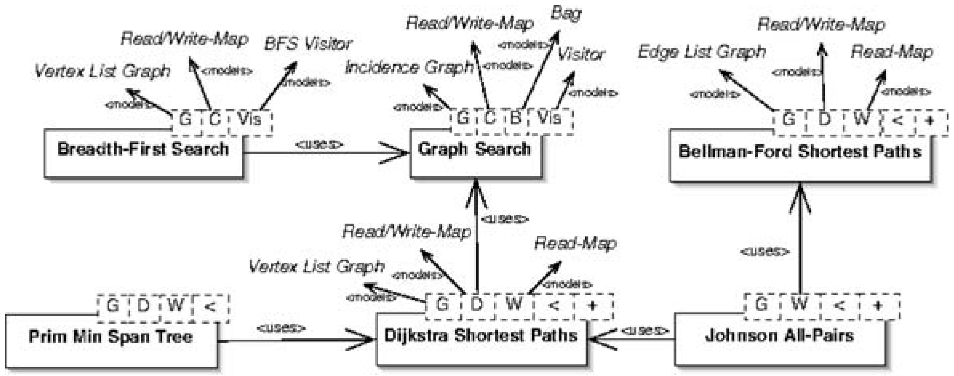

Fig. 3. Graph algorithm parameterization and reuse within the graph library. Arrows for redundant models relationships are not shown. For example, the type parameter ***G*** of breadth-first search must also model Incidence Graph because breadth-first search uses graph search.

Figure 3 depicts the graph algorithms, how they are related, and how each is parameterized. Each large box represents an algorithm and the attached small boxes represent type parameters. An arrow labeled <***uses***> from one algorithm to another specifies that one algorithm is implemented using the other. An arrow labeled <***models***> 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 has 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. Furthermore, breadth-first search is implemented using the graph search algorithm. In order to minimize clutter, Figure 3, as indicated in its caption, does not show redundant models relationships.

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 parameter ***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 (which is not to be confused with the Visitor design pattern). The graph search algorithm also takes a type parameter ***C*** for mapping each vertex to a color, a means for keeping track of which vertices have been visited during the search. The type parameter ***C*** must 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; and have values associated with their vertices and edges. For example, a graph might depict the layout of a communication network, its vertices representing endpoints and its edges representing direct links. 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 either representation.

The graph algorithms are parameterized on the graph type. Breadth-first search takes a 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 and Vertex List Graph defines an interface for accessing the vertices of a graph in an unspecified order. The Bellman-Ford shortest paths algorithm (Bellman, 1958) requires a model of the Edge List Graph concept, which provides access to all the edges of a graph.

The partitioning of the graph capabilities into three concepts illustrates generic programming's emphasis on minimal algorithm requirements. The Bellman-Ford shortest paths algorithm requires of a graph only the operations described by the Edge List Graph concept. Breadth-first search, in contrast, requires the functionality of two separate concepts. By partitioning the functionality of graphs, each algorithm can be used with any data type that meets its minimum requirements. If the three fine-grained graph concepts were replaced with one monolithic concept, each algorithm would require more from its graph type parameter than necessary and would thus 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 its implementation involves 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. Furthermore, 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 example use of the algorithm.

4 Graph Library in C++

C++ generics were intentionally designed to exceed what is required to implement type-safe polymorphic containers. The resulting template system provides a platform for experimentation with, and insight into the expressive power of, generic programming. Before templates were added to the language, 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 (Stroustrup, 1994). 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 stand-alone 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, C# generics, 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 required in practice (Stroustrup, 1994).

Two C++ language features combine to enable generic programming: templates and function overloading. C++ provides both function templates and class templates, but function templates are more central to generic programming.

In C++, templates are not separately type checked. Instead, type checking is performed after instantiation at each call site. Type checking of the call site can only succeed when the input types satisfy the type requirements of the function template body. Unfortunately, because of this, if a generic algorithm is invoked with an improper type, complex, long, and potentially misleading error messages may result.

```
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 == graph_traits<G>::vertex
//   map_traits<C>::value models Color
//   Vis models BFS Visitor
```

Fig. 4. Breadth-first search as a function template.

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

Fig. 5. Sketch of a concrete graph implementation.

4.1 Implementation

We implement the breadth-first search algorithm as the ***breadth_first_search*** function template, declared in Figure 4. C++ does not provide direct support for constraining type parameters; standard practice is to express constraints in documentation in conjunction with meaningful template parameter names (Austern, 1998). Techniques for checking constraints in C++ exist in library form (Siek & Lumsdaine, 2000; McNamara & Smaragdakis, 2000). These techniques, however, do not constitute actual language support and involve the addition of compile-time assertions to the bodies of generic algorithms.

The ***graph_traits*** class template used in Figure 4 provides access to the associated types of the graph type. In particular, the vertex type is referenced using the type expression ***graph_traits<G>::vertex***. Traits classes are an idiom used in C++ to map types to other types or functions (Myers, 1995). 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 type definitions and member functions. Figure 5 shows a specialization of ***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 standard function overloading rules for C++. Each call is resolved to the most specific overload for the types of its arguments.

Table 4. *Documentation for the graph concepts. Object **g** is of type **G**, **e** is of type **edge**, and **v** is of type **vertex***

Graph concept	
graph_traits<G>::vertex	
graph_traits<G>::edge	
src(e, g)	***vertex***
tgt(e, g)	***vertex***
Incidence Graph refines Graph	
graph_traits<G>::out_edge_iter models Iterator	
The value type of ***out_edge_iter*** is ***edge***.	
out_edges(v, g)	***pair<out_edge_iter>***
out_degree(v, g)	***int***
Vertex List Graph	
graph_traits<G>::vertex_iter models Iterator	
The value type of ***vertex_iter*** is ***vertex***.	
vertices(g)	***pair<vertex_iter>***
num_vertices(g)	***int***

Documentation for the graph concepts is shown in Table 4. In addition to valid expressions, concept documentation specifies how to access associated types such as the vertex, edge, and iterator types using the ***graph_traits*** class, same-type constraints, and refinement relationships.

A sketch of a concrete adjacency list implementation is shown in Figure 5. The ***AdjacencyList*** class models the Incidence Graph and Vertex List Graph concepts, but since C++ has no mechanism for specifying models, these relationships are implicit. The ***graph_traits*** class is specialized for ***AdjacencyList*** so its associated types can be accessed within function templates.

The Read/Write Map and Read Map concepts are defined in Table 5; the BFS Visitor concept is defined in Table 6.

The code below presents an example use of the ***breadth_first_search*** function to output vertices in breadth-first order:

```
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;
```

Table 5. *Documentation for the property map concepts. Object **m** is of type **M**, object **k** is of type **key**, and **v** is of type **value***

Read Map concept	
map_traits<M>::key	
map_traits<M>::value	
get(m, k)	***value***

Read/Write Map concept refines Read Map
put(m, k, v)

Table 6. *Documentation for the* BFS Visitor *concept. Object **vis** is of type **V**, **g** is of type **G**, **e** is of type **graph_traits<G>::edge**, and **v** is of type **graph_traits<G>::vertex***

BFS Visitor
vis.discover_vertex(v, g)
vis.finish_vertex(v, g)
vis.examine_edge(e, g)
vis.tree_edge(e, g)
vis.non_tree_edge(e, g)
vis.gray_target(e, g)
vis.black_target(e, g)

```
    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 0;
}
```

The ***test_vis*** visitor overrides the ***discover_vertex*** function: empty implementations of the other visitor functions are provided by ***default_bfs_visitor***. A graph is constructed using the ***AdjacencyList*** class, and then ***breadth_first_search*** is called. The body of the ***breadth_first_search*** function template is type checked at its call site. 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 not be consistent with the documented constraints for the function).

4.2 Evaluation of C++

C++ templates enable the expression of generic algorithms, even for large and complex generic libraries. It is relatively easy to convert concrete functions to function templates, and since they support implicit argument deduction, calling function templates does not syntactically differ from calling non-generic functions. The traits mechanism provides an effective way to access associated types. Since C++ uses structural matching to type check generic function calls, retroactive modeling is automatic (see Section 12.4) and constraints on associated types and type parameters are implicit. Multi-type concepts are easily accommodated in this model. Finally, complex type names can be aliased to simpler names using the ***typedef*** keyword.

The C++ template mechanism has drawbacks, however, with respect to modularity. The complete implementations of templates reside in header files (or an equivalent). Thus, code that calls a template function must be recompiled if the template implementation changes. 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 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.

5 Graph library in Standard ML

To implement a generic library in Standard ML we 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 specifies a type for each value and a signature for each nested structure. In essence, signatures play the same role for structures that types play for values. Functors are templates for creating new structures and can be parameterized on values, types, and structures. Multiple structures of similar form can be represented using a single functor that captures characteristics the structures hold in common. Differences between these structures are represented by the functor's parameters. The following discussion demonstrates how structures, signatures, and functors together enable generic programming.

5.1 Implementation

We express concepts in SML using signatures. The following code shows SML 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 out_degree : graph_t → vertex_t → int
    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. Additionally, ***vertex_t*** must be an ***equality type***, meaning ***vertex_t*** values must be comparable using the equality (=) operator.

IncidenceGraphSig and ***VertexListGraphSig*** both demonstrate how to express concept refinement in SML. 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 many other signatures, each type or value must be specified only once, either in one of the included signatures or in the body of the signature that is being defined. As illustrated shortly, this limitation has negative implications for generic programming.

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

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

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

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

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

    fun edges (Data 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 ***AdjacencyList*** 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, ***AdjacencyList*** is said to model the Graph, Vertex List Graph, and Incidence Graph concepts. ***AdjacencyList*** also defines 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 SML, we can implement algorithms using functors. The following code illustrates the general structure of a generic breadth-first search implementation:

```
  functor MakeBFS(
    structure G1 : IncidenceGraphSig
    structure G2 : VertexListGraphSig
    structure C : ColorMapSig
    structure Vis : BFSVisitorSig
    sharing G1 = G2 = Vis
    sharing type C.key_t = G1.vertex_t
  ) = 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 arguments that satisfy certain requirements, it creates a new structure specialized for the functor parameters. A functor's parameters are specified using the same syntax as the body of a signature. This notation is convenient for expressing constraints on those parameters. The ***MakeBFS*** functor takes four structures as parameters and enforces concept requirements by constraining the structures with signatures. The ***G1*** structure, for example, is constrained by the ***IncidenceGraphSig*** signature.

Our generic breadth-first search algorithm can be applied to any single graph type that models two concepts: Vertex List Graph and Incidence Graph. Generic algorithms in SML are parameterized to take structures that must match a specific signature. Thus, breadth-first search in SML should ideally take one graph structure whose signature combines the constraints of the two required graph concept signatures.

Unfortunately it is not always possible to combine two existing signatures as desired. For example, since signatures can be constructed inline, one might attempt to write:

```
(* ERROR: VertexListGraphSig and IncidenceGraphSig overlap *)
functor MakeBFS(
  structure G : sig include IncidenceGraphSig
                    include VertexListGraphSig
                end
  ...) = ...
```

to express that structure ***G*** conforms to both interfaces; but this does not work because ***VertexListGraphSig*** and ***IncidenceGraphSig*** overlap: both signatures define the ***graph_t*** and ***vertex_t*** types, which they included from the ***GraphSig*** signature. However, this technique works for signatures that do not share identifiers.

Since the necessary signature cannot be constructed programmatically, a signature ***VertexListAndIncidenceGraphSig*** to combine the identifiers from the original signatures could be implemented by hand. Doing so, however, imposes the burden of manually keeping the related signatures consistent.

The method we used to implement breadth-first search differs from the techniques just described. Instead, 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. When the ***MakeBFS*** functor is applied, the same structure is used for both arguments, thereby providing the two necessary views on the same model, at the cost of an extra functor parameter.

In addition to listing required structures, ***MakeBFS*** specifies that some type names in the structures must refer to identical types. These are denoted as ***sharing specifications***. Two sharing specifications appear in the ***MakeBFS*** definition. The first is a structure sharing specification 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, both ***G1*** and ***G2*** are required by their signatures to contain a nested type ***vertex_t***. The sharing specifies that ***G1.vertex_t*** and ***G2.vertex_t*** must refer to the same type. The second sharing, a type sharing specification, declares that ***C.key_t*** and ***G1.vertex_t*** must be the same as well. In addition to the requirements they express using concepts, algorithms require certain relationships between their type arguments. Sharing specifications capture and enforce those relationships.

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

```
structure BFS =
MakeBFS(structure G1 = AdjacencyList
        structure G2 = AdjacencyList
        structure C = ALGColorMap
        structure Vis = VisitImpl)

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

First, the algorithm is instantiated by applying ***MakeBFS*** to a set of structures that meet the ***MakeBFS*** requirements. The ***AdjacencyList*** structure is assigned to both the ***G1*** and ***G2*** parameters, matching the ***IncidenceGraphSig*** and ***VertexListGraphSig*** signatures respectively. Though awkward, this formulation avoids the explicit definition of a ***VertexListAndIncidenceGraphSig*** signature. 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 specified above. Application of the ***MakeBFS*** functor realizes the ***BFS*** structure, which encapsulates a ***breadth_first_search*** function specialized with the above structures. Finally, ***BFS.breadth_first_search*** is called with arguments that match the now concrete type requirements.

5.2 *Evaluation of Standard ML*

SML signatures and structures conveniently express concepts and concept models using nested types to represent both the modeled types and associated types and using functions to implement valid expressions. A multi-type concept simply has more than one type that is considered primary. Associated types can be required to model a concept using nested structures and sharing specifications within the concept signature. The structure representation of concept models promotes 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.

In SML, long and complicated type names can be abbreviated using the ***type*** mechanism. Since SML structures can implicitly conform to signatures, retroactive modeling is automatic: no mechanism is needed to tie a new signature to an old structure. Thus, a generic SML algorithm, written in terms of signatures, can operate on any structures that meet its requirements. Unfortunately, a generic algorithm's representative functor must be explicitly instantiated before calling the function.

Standard ML does not support separate compilation of modules in general; however, generic algorithms specified as described here can be type checked and compiled independent of calls to them. All the parameters to the functor that defines a generic algorithm can be related using sharing specifications. The functor body can then be type checked with respect to the sharing specifications, independent of any application of the functor. Sharing specifications express the constraints imposed by an algorithm on its arguments that are not imposed by concepts. All necessary sharing relationships between functor parameters must be declared explicitly. If not, the type checker will issue type checking errors when the functor is analyzed. When a functor is applied, SML verifies that the functor's arguments meet the sharing and signature requirements.

A language that supports generic programming should facilitate modularity by statically checking models against their concepts, independent of their use with generic algorithms. To do this in SML, a structure's definition 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. However, 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 ***AdjacencyList*** structure were declared:

```
structure AdjacencyList : 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 ***AdjacencyList*** to the necessary signatures can be checked as shown in the following code outline:

```
structure AdjacencyList = struct ... end

structure AdjacencyListCheck1 : IncidenceGraphSig = AdjacencyList;
structure AdjacencyListCheck2 : VertexListGraphSig = AdjacencyList;
```

The structures ***AdjacencyListCheck1*** and ***AdjacencyListCheck2*** are assigned ***AdjacencyList*** and constrained by the ***IncidenceGraphSig*** and ***VertexListGraphSig*** signatures respectively. Each of these structures confirms statically that ***AdjacencyList*** conforms to the corresponding signature without limiting access to its structure members. This technique as a side effect introduces the unused ***AdjacencyListCheck1*** and ***AdjacencyListCheck2*** structures.

As previously described, the ***include*** mechanism for signature combination in SML can often be used to implement a concept that refines another concept or to express multiple constraints on a parameter to an algorithm; but this mechanism does not work if any of the concepts involved overlap. SML programs sometimes use structure containment to approximate signature combination. For example, the Vertex List Graph and Incidence Graph concepts might have been combined as follows:

```
signature VertexListAndIncidenceGraphSig = sig
  structure VLG : VertexListGraphSig
  structure IG : IncidenceGraphSig
  sharing VLG = IG
end
```

Rather than attempting to ***include*** the signatures for the above two concepts, this signature contains them and applies a sharing specification that requires all shared structure between the two to be the same. A generic algorithm could then be written in terms of the above signature.

This idiom for combining signatures is roughly equivalent to passing two separate instances of the same structure to an algorithm, the mechanism that we used above. This combination, however makes a somewhat different tradeoff. For instance, the breadth-first search algorithm could be specified as follows:

```
functor MakeBFS(
  structure G : VertexListAndIncidenceGraph
  structure C : ColorMapSig
  structure Vis : BFSVisitorSig
  sharing G.G1 = Vis
  sharing type C.key_t = G.G1.vertex_t
) = struct
  structure G1 : IncidenceGraph = G.IG
  structure G2 : VertexListGraphSig = G.VG
```

```
    fun breadth_first_search g v vis map = ...
  end;
```

Here, the ***MakeBFS*** functor takes a ***VertexListAndIncidenceGraphSig*** structure and internally decomposes the combination. The algorithm could then be instantiated as follows:

```
structure BFS =
  MakeBFS(struct
                structure G1 = AdjacencyList
                structure G2 = AdjacencyList
             end
             structure C = ALGColorMap
             structure Vis = VisitImpl)
```

Here, a new structure is created inline to model the combined concepts. One advantage of this formulation is that the inline structure used above could be created out-of-line and used to instantiate other algorithms. Furthermore, the signature encapsulates the sharing specification between the two instances of the structure in one place. This style, however, requires the implementation of new structures for every combination of graph concepts required by algorithms. The extra syntax used to decompose the structure within an algorithm cancels out the syntactic abbreviation used in the functor parameter list. Ideally, SML would specify an algebra of signatures by which multiple signatures could be combined to form a single new signature. An ML-style module system is extended in (Ramsey *et al.*, 2005) to provide an expressive signature language that supports composition and other useful operations on signatures. Using this extension, the breadth-first search algorithm appears as follows:

```
functor MakeBFS(
  structure G : VertexListGraph andalso IncidenceGraph
  ...) = struct ... end
```

This formulation clearly expresses that the structure ***G*** must meet the constraints of both ***VertexListGraph*** and ***IncidenceGraph***. The signature language semantics can coalesce the shared structure of these signatures because they meet the language extension's notion of compatibility.

Functors are not the only means for implementing generic algorithms. SML 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, often called a ***dictionary***. The concept's associated types become type parameters to the data type, and its valid expressions become the dictionary functions. In addition to other arguments, 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 SML, though possible, is not ideal. In larger SML programs, managing dictionaries manually becomes cumbersome and increases the code base significantly. This situation is analogous to manually implementing virtual function tables in C rather than leveraging the object-oriented programming features of C++. In fact, some Haskell implementations effectively translate programs that use generics (type classes) to equivalent Haskell programs expressed in this dictionary-passing style. Automating the mechanisms of generic programming is preferable to implementing them manually.

6 Graph library in OCaml

Objective Caml (Leroy, 2000) comes from the ML family of programming languages. It has a functional core, much like Standard ML, augmented with a novel and flexible object system (Rémy & Vouillon, 1997). The OCaml module system is semantically close to that of Standard ML and can support the same functor-driven generic programming style with the same level of effectiveness (OCaml adds support for higher-order functors, but their effect on generic programming is minimal). However, the OCaml object system enables another style of generic programming and we explore that alternate style in this section.

OCaml's object system differs markedly from those of the other object-oriented languages under study, especially in that it explicitly distinguishes ***classes*** from ***object types***. A class is a blueprint for creating objects. It determines the data that is private to an object and defines the behavior that an object supports. Object types are the types assigned to objects by the type system; an object type is basically a set of method signatures. The following code is an example of a class definition in OCaml.

```
class myclass = object
  method m x = x + 7
end
```

This definition accomplishes three tasks. First, the name ***myclass*** is bound to the class created by the class expression ***object ... end***. Second, the name ***myclass*** becomes a synonym for the structural object type <***m : int*** → ***int***>. This contrasts with most object-oriented languages where each class is a new unique type. Overloading the name ***myclass*** for both the class and the object type is not a problem because the two meanings are used in different contexts. Finally, once a class has been declared, objects of that class can be created, and its methods called, as in the following:

```
let x = new myclass in
  x#m 35
```

The ***new*** expression creates a ***myclass*** object; the ***m*** method is invoked on object ***x*** using the hash (#) notation.

Object types may be used to declare the type of a parameter to a function. For example, consider the following:

```
let call_m (x : <m : int → int>) = x#m 35
```

Here <***m : int*** → ***int***> is the type of the parameter ***x***. The ***call_m*** function accepts as its argument any object whose class provides exactly one public function named ***m*** that when applied to a value of type ***int*** returns a value of type ***int***. An object of class ***myclass*** is an acceptable argument to this function, but so is an object of the following class:

```
class anotherclass = object
  method m a = 35 + a
end
```

Once declared, the class names ***myclass*** and ***anotherclass*** both serve as abbreviations for the object type <***m : int*** → ***int***> and may be used in its place, for example:

```
let call_m (x : myclass) = x#m 35
```

The use of the class name as an abbreviation for an object type somewhat blurs the distinction between classes and object types for the sake of usability. Object types provide a structural means for limiting the possible arguments to a function. This differs from most mainstream object-oriented languages, in which the types for classes with similar structure but different names are always distinct.

The ***call_m*** function defined above is more restrictive than one might desire because it accepts only objects whose class contains exactly one function named ***m***. To achieve more flexibility, OCaml has open object types that match any object type containing at least the required methods and possibly more methods. For example, consider the following function:

```
let call_m2 (x : <m : int → int; ..>) = x#m 35
```

The object type of parameter ***x*** is open, denoted by **..**, so it can match with a type such as ***yourclass***.

```
class yourclass z = object
  method m (x:int) = x + z
  method n (y:int) = y + z
end

let y = new yourclass 7 in
  call_m2 y
```

A class (or class type) name prefixed with # denotes the corresponding open type. So the following definition of ***call_m2*** function is equivalent to the one above.

```
let call_m2 (x : #mytype) = x#m 35
```

The underlying mechanism that allows ***y*** to be passed to ***call_m2*** is not subtyping polymorphism, as is typical for most object-oriented languages, but instead parametric polymorphism. A function such as ***call_m2*** with a parameter that has an open type is implicitly a polymorphic function. There is a hidden parameter, called a ***row variable***, associated with the **..** of the open type. The reliance on parametric polymorphism instead of subtyping leads to many subtle and important differences from other object-oriented languages. For example, it allows OCaml to easily deal with programming situations where binary methods or virtual types are needed (Leroy *et al.*, 2003; Rémy & Vouillon, 1998), situations that are problematic for most object-oriented languages.

OCaml provides ***class types*** as a means to conveniently define, name, and combine object types. Their form is similar to that of class expressions, though the definition starts with the syntax ***class type*** and the methods are replaced with signatures. For example, the definition:

```
class type mytype = object
  method m : int → int
end
```

introduces the name ***mytype*** as an abbreviation for the object type <***m : int*** → ***int***>. Using this definition, the ***call_m*** function can be rewritten as follows:

```
let call_m (x : mytype) = x#m 35
```

In addition to serving as abbreviations, class types can be combined to establish new class types. Consider the following two class types:

```
class type onetype = object
  onemethod : unit
  anothermethod : unit
end

class type twotype = object
  anothermethod : unit
  yetanothermethod : unit
end
```

Each class type describes an object type. Under some circumstances, it may be desirable to describe a class that combines the functionality of both of the above class types. Then one may duplicate the information above as in the following:

```
class type bigtype = object
  onemethod : unit
  anothermethod : unit
  yetanothermethod : unit
end
```

Alternatively, one may describe the above class type using the ***inherit*** syntax, as in the following:

```
class type bigtype = object
  inherit onetype
  inherit twotype
end
```

The ***inherit*** syntax imports the members of the named class type into the class type being defined. In the case of multiple ***inherit*** statements, the method names declared in the inherited class types must be compatible. In the above example, both ***onetype*** and ***twotype*** declare methods named ***anothermethod***. The two declarations of ***anothermethod*** are identical, so they are unified for the definition of ***bigtype*** (Standard ML signatures do not support such combination; see Section 5.1).

6.1 Implementation

In OCaml, we represent concepts as object types, expressed using class types. The following code shows OCaml representations of graph concepts for the breadth-first search algorithm:

```
(* Vertex List Graph concept *)
class type ['vertex_t] vertex_list_graph = object
  method vertices : 'vertex_t list
  method num_vertices : int
end

(* Edge concept *)
class type ['vertex_t] edge = object
  method source : 'vertex_t
  method target : 'vertex_t
end

(* Incidence Graph concept *)
class type ['vertex_t, 'edge_t] incidence_graph = object
  constraint 'edge_t = 'vertex_t #edge
  method out_edges : 'vertex_t → 'edge_t list
end
```

Similarly to other object-oriented languages under study, OCaml exhibits some limitations with respect to the representation of associated types. Specifically, OCaml does not provide a concise mechanism for mappings from types to types, such as nested type names in C++ classes and ML structures. Instead, associated types are expressed as type parameters. For example, the ***incidence_graph*** class type captures two associated types, ***'vertex_t*** and ***'edge_t***, as type parameters. Associated types expressed in this manner contribute to cluttered syntax because every associated type must be named in every context where the concept is used, regardless of the associated type's relevance. If a concept has many associated types, the resulting code may be tedious to write and difficult to read. However, OCaml does have a mechanism for expressing constraints on associated types. We use a ***constraint*** clause in the ***incidence_graph*** class type, so that the ***'edge_t*** type is required to meet the ***edge*** interface. We discuss this issue in more detail in Section 12.1.

Valid expressions are represented using class methods. Class methods can be polymorphic and recursively typed. Where other object-oriented languages provide member function templates, which are member functions parameterized on inheritance-constrained types, OCaml methods can have polymorphically typed parameters. For the purpose of generic programming, either is adequate.

Like the other object-oriented languages under study, models of concepts are represented using classes. The following code implements an adjacency-list based graph type that models the Incidence Graph and Vertex List Graph concepts:

```
class algraph_edge (s:int) (t:int) = object
  val src = s
  val tgt = t
  method source = src
  method target = tgt
end

class adjacency_list (num_vertices_) = object
  val g = Array.make num_vertices_ []

  method add_edge (src,tgt : int*int) = g.(src)<-(tgt::g.(src))

  method vertices =
    let rec floop (i:int) =
      if i = num_vertices_ then [] else i::floop (i+1)
    in floop 0

  method num_vertices = num_vertices_
  method out_edges v = List.map (fun n → new algraph_edge v n) g.(v)

  method adjacent_vertices v = g.(v)

  method edges =
    let (_,result) = Array.fold_left
        (fun (src,(sofar:(algraph_edge) list)) (tgts:int list) →
          (src+1, List.append
              (List.map (fun n → new algraph_edge src n)
                 tgts) sofar))
        (0,[]) g in result

  method create_property_map : 'a. 'a → 'a array =
    fun def → Array.make num_vertices_ def
end
```

Algorithms are expressed as polymorphic functions constrained by open object types. The following code shows the signature of the graph search algorithm as implemented in OCaml:

```
let graph_search
(graph : (('vertex_t,'edge_t) #incidence_graph) as 'graph_t)
(v : 'vertex_t)
(q : 'value_t #buffer)
(vis : ('graph_t, 'vertex_t, 'edge_t) #visitor)
(map : ('color_t, 'key_t) #color_map) = ...
```

The parameters to the algorithm are qualified by the open variants of the object types. These types limit valid arguments to only those that structurally meet the class type requirements.

In Standard ML, sharing specifications explicitly specify that certain associated types are the same. To express the same constraints in OCaml, the type variable name for each associated type is shared among the relevant object types. For example, the type variables ***'vertex_t*** and ***'edge_t*** in the above code name parameters to multiple class types. The function will thus type check only for combinations of classes whose object types match the expansions of the object types specified above. Furthermore, the complex type of the ***graph*** parameter is given the name ***graph_t***, using the ***as*** syntax, and ***graph_t*** is subsequently used to constrain the ***vis*** parameter.

6.2 Evaluation of OCaml

The combination of object types and row variable polymorphism conveniently expresses the concepts present in the graph library. As with the other object-oriented languages, OCaml provides limited support for associated types. They must be represented as parameters to a class interface and as such they must all be named wherever an object type is referenced. However, OCaml provides a mechanism for constraining associated types using ***constraint*** clauses. Section 12.1 discusses problems related to constraining associated types in Java and C#.

OCaml does not support multi-type concepts, for object types and objects are the units of generic programming. As with classes in the more mainstream object-oriented languages, an object type constrains only one primary type.

OCaml provides no convenient means to constrain an algorithm parameter with more than one concept. As a consequence, a special-case class type must be introduced whenever an algorithm parameter must model multiple concepts. For example, consider the following code:

```
class type ['vertex_t, 'edge_t] vertex_list_and_incidence_graph = object
  inherit ['vertex_t] vertex_list_graph
  inherit ['vertex_t, 'edge_t] incidence_graph
end
```

This class type captures both the Vertex List Graph and Incidence Graph concepts. In essence, it defines the Vertex List and Incidence Graph concept as a refinement of the two component concepts. This technique works but it introduces a possible combinatoric explosion of uninteresting concepts that simply combine meaningful concepts without adding meaning themselves.

Object types define only the structural properties of objects, so retroactive modeling is merely a matter of structural conformance. Structural conformance gives retroactive modeling for free.

OCaml supports type aliases with its ***type*** form. Complex and long type names can be abbreviated using this mechanism. Furthermore, the ***as*** form is a convenient means to define type aliases within function parameter lists for type expressions that are used as constraints. The ***graph_search*** algorithm definition shows an example use of the ***as*** form.

OCaml functions need not be explicitly instantiated. Object types capture function requirements in a manner compatible with traditional parametric polymorphism.

Objects are our models, so the arguments to an algorithm carry their valid operations with them as methods.

With respect to support for separate compilation of generic algorithms and models, our implementation of the graph library brought to light a rather striking property of the OCaml type system. Open type annotations on function parameters participate symmetrically in the type inference process. The constraints specified on type parameters may not completely determine the externally-visible type of the function. Consider the following valid code:

```
class type conceptA = object
  method fn1 : int
end
let algo (x : #conceptA) : int = x#fn2 + 1
```

It declares a class type ***conceptA*** that has one method signature ***fn1***. The function ***algo*** accepts one parameter that is constrained by ***conceptA*** and returns an ***int***. The body of ***algo*** contains a call to the ***fn2*** method of ***x***. The algorithm passes type checking, but the inferred type is <***fn1 : int, fn2 : int, ..***> → ***int*** rather than <***fn1 : int, ..***> → ***int***. This occurs because the row variable in the ***#conceptA*** object type unifies with the inferred type of the function body, yielding a type definition that accurately expresses the type of the body. Open type annotations on function arguments cannot limit the operations in terms of which the function body is defined. The difference between the type annotations and the inferred type may go unnoticed by a library developer and thereby place undue burden on library users. In order to avoid this issue, an algorithm implementor may initially constrain parameters with closed types, only replacing them with open variants after the implementation is complete. For example the ***algo*** function rewritten as follows would fail to type check and thereby alert the implementor that the function body exceeds the set of operations allowed by the type annotations.

```
  let algo (x : conceptA) : int = x#fn2 + 1
```

Of course, adding and removing #'s is a tedious business.

Finally, OCaml, like Standard ML, does not provide a mechanism for verifying that a class models a concept without introducing additional class names or hiding unrelated class functionality. Effective use of the module system to hide undesirable names might limit the visibility of any unwanted class names.

7 Graph library in Haskell

The Haskell community uses the term "generic" to describe a form of generative programming with respect to algebraic datatypes (Backhouse *et al*., 1999; Hinze & Jeuring, 2003; Jeuring & Jansson, 1996). 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, 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.

7.1 Implementation

We express the graph library concepts in Haskell with the following type classes:

```
class Edge e v | e → v where
  src :: e → v
  tgt :: e → v

class (Edge e v) ⇒ IncidenceGraph g e v | g → e, g → v where
  out_edges :: v → g → [e]
  out_degree :: v → g → Int

class (IncidenceGraph g e v) ⇒ BidirectionalGraph g e v where
  in_edges :: v → g → [e]
  in_degree :: v → g → Int
  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, called subclassing, is the Haskell mechanism for concept refinement and for placing constraints on associated types. For example, the Bidirectional Graph concept is a refinement of the Incidence Graph concept and the Incidence Graph concept requires its edge type to model Edge.

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. This is analogous to how the object-oriented languages under study implement associated types, including Objective Caml. 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 the Haskell 98 standard, type classes are restricted to a single parameter, but many Haskell implementations support multiple parameters. The syntax ***g*** → ***e*** denotes a functional dependency, another extension to Haskell (Jones, 2000; Peyton Jones *et al.*, 1997). The meaning of ***g*** → ***e*** is that for a given graph type ***g*** there is must be unique edge type ***e***. Without functional dependencies it would be difficult to construct a legal type in Haskell for ***breadth_first_search***. The problem is that the ***IncidenceGraph*** type class has three parameters, including an edge parameter ***e***.

However, ***breadth_first_search*** does not include an actual parameter that is an edge, so the type system would not be able to deduce the argument type for ***e***. With the functional dependency ***g*** → ***e***, the type system can deduce the type argument for ***e*** from the type argument for ***g*** and the instance declarations. In Section 12.3 we discuss how the lack of functional dependencies (or an equivalent mechanism) forces programmers to write explicit instantiations in languages such as C# and Cecil.

The ***BFSVisitor*** type class, shown below, is parameterized on the graph, queue, and output types. The queue and output types are needed because Haskell is a pure functional language and any state changes must be passed around explicitly, as is done here, or implicitly using monads. The ***BFSVisitor*** concept is also parameterized on the vertex and edge types because they are associated types of the graph. The ***BFSVisitor*** type class has default implementations of its valid expressions that do nothing.

```
class (Edge e v) ⇒ BFSVisitor vis q a g e v | g → v, g → e 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, as well as the explicit ***instance*** declarations to establish that ***AdjacencyList*** is a model of the concepts Incidence Graph and Vertex List Graph, are shown below:

```
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 Edge Edge Vertex where
  src (E s t) = V s
  tgt (E s t) = 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
```

The Haskell signature for the breadth-first search function is shown below. The first line gives the name, and the following two lines give the context of the function which expresses constraints on the type parameters. Haskell can infer constraints, so we could leave out the context, but we prefer to make the constraints explicit to provide documentation and check that the documentation is correct. The

breadth_first_search 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 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 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.

7.2 *Evaluation of Haskell*

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. However, the support for associated types in Haskell is less than ideal. The addition of associated types to 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). In response to our earlier paper (Garcia *et al.*, 2003), a proposal was formulated to add direct support for associated types to Haskell type classes, first in the form of nested datatypes (Chakravarty *et al.*, 2005b), and later also in the form of nested type synonyms (Chakravarty *et al.*, 2005a).

Haskell provides the ability to place constraints on associated types via subclassing. An example of this is in the ***IncidenceGraph*** type class, which constrains the edge type to be an instance of the ***Edge*** type class.

Retroactive modeling is provided in Haskell by ***instance*** declarations: ***instance*** declarations are separate from type definitions. Haskell supports type aliases. In fact, type aliases can be parameterized, which we found to be useful in the graph library implementation. 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 tend to be difficult to understand. We believe this is because the Haskell type system is based on type inference. 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. Recent work targets this issue and promising results have been achieved (Haack & Wells, 2003). We developed the graph library initially with the Hugs release from November 2002 and we later experimented with GHC 6.2. The error messages from GHC were a considerable improvement over Hugs, but the issue described above was still a problem.

In Haskell, invoking a polymorphic function is almost as easy as invoking a non-generic function. The type arguments for the polymorphic function are implicitly deduced as part of Haskell's type inference system. However, in some cases users of a polymorphic function must do extra work to declare their types to be instances of the type classes that are 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.

8 Graph Library in Eiffel

Eiffel supports generics through parameterized classes; formal type parameters follow the class name within square brackets. The design of the graph library leverages Eiffel's *conformance* relation (Eiffel's term for substitutability) to express constraints on the type parameters to generic algorithms. In Eiffel, each type parameter can be accompanied by a constraint, a type to which the actual type argument must conform. Syntactically, the arrow (→) symbol attaches a constraint to a type parameter. If a constraint is omitted, it defaults to the ***ANY*** class, the root of the Eiffel class hierarchy. The constraining type may refer to other type parameters.

8.1 Implementation

Concepts are represented as deferred classes (cf. abstract classes in C++). Figure 6 shows the implementations of three graph concepts. Eiffel classes offer no direct mechanism for attaching types to classes; thus, associated types are expressed as type parameters, as in OCaml and Haskell. The ***V*** and ***E*** parameters are examples of this. Concept refinement is represented using inheritance between deferred classes; the ***VERTEX_LIST_AND_INCIDENCE_GRAPH*** class demonstrates refinement. To model a concept, a type inherits from the class representing the concept; the ***ADJACENCY_LIST*** class is an example of this.

The interface of the breadth-first search algorithm, shown in Figure 7, is representative of the generic algorithms in the Eiffel graph library implementation.

```
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]]
   ...
```

Fig. 6. Two graph concepts and a class that conforms to these concepts. ***V*** and ***E*** stand for the vertex type and edge type, respectively.

```
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 ...
```

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

Eiffel does not support parameterized methods. Therefore, generic algorithms are implemented as parameterized classes with a single method, which we chose to name ***go***. The graph type ***G*** is required to model the Vertex List Graph and Incidence Graph concepts. The combined set of requirements of these two concepts is included in the class ***VERTEX_LIST_AND_INCIDENCE_GRAPH[V, E]***, shown in Figure 6. This combination class is necessary because Eiffel does not currently support constraining a single type parameter with multiple classes. Constraints frequently refer to associated types, such as the vertex and edge types. Since associated types are type parameters of the concept classes, a generic algorithm must have a type parameter for each distinct associated type of any concept that constrains that algorithm.

The graph library implementation in Eiffel necessarily deviates from the design described in Figure 3 in that it 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 encapsulate associated types, are discussed below. Furthermore, this section discusses how Eiffel's support for covariant type parameters can lead to the compiler rejecting reasonable code as potentially unsafe. This was encountered frequently with the original library design. Unfortunately, with fewer type parameters, not all of the exact types of the arguments or return values of generic algorithms can be expressed. For example, in Figure 6 the return type of the ***vertices*** method 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, since exact types are not exploited for static dispatching. However, inexact types can result in situations where either downcasting or relying on covariance in type parameters is needed.

8.2 Evaluation of Eiffel

Among object-oriented languages, Eiffel was an early provider of constrained generics. In Eiffel, generic classes are type checked independent of their uses, allowing type errors to be caught before a generic class is instantiated, and enabling separate compilation of generic classes. We found, however, that notable difficulties follow from other design choices of Eiffel, in particular from (1) the need to explicitly instantiate generic algorithms, (2) the mechanisms for representing and accessing associated types, (3) the lack of support for multiple constraints on a single type parameter, and (4) allowing covariant change in type parameters.

Eiffel requires explicit instantiation: the caller of a generic algorithm must provide both the actual function arguments and the type arguments. Assuming the function 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 makes calls to generic functions verbose, particularly when the function has many type parameters. In the above example, with three type parameters, the type parameter list is already much longer than the entire function call. In addition, Eiffel lacks static methods, and thus an object must be explicitly created before the algorithm can be invoked. Another effect of explicit instantiation is that calls to generic algorithms often carry unnecessary dependencies on the actual types of the function arguments. For example, changing the graph object ***g*** to some other compatible graph type requires changing all ***bfs*** call sites that use ***g***, because they explicitly mention ***g***'s type. Section 12.3 discusses the effect of explicit instantiation in detail.

Associated types and constraints imposed on them are part of concept requirements but cannot be properly encapsulated with Eiffel classes. Instead, every use of a concept as a constraint must repeat all of its associated types and their constraints. The constraint ***VERTEX_LIST_AND_INCIDENCE_GRAPH[V, E]*** in Figure 7, for example, must mention the type parameters ***V*** and ***E*** that correspond to the associated vertex and edge types. This example does not demonstrate repetition of constraints, as none are placed on ***V*** and ***E***. Examples of constraint repetition are postponed until Section 12.1, which discusses this common phenomenon in Eiffel, Java, and C#. Eiffel's *Anchored types* (Meyer, 1992) do not provide a means to encapsulate associated types because anchored types cannot express arbitrary dependencies between types.

Another set of problems arises as the combined effect of explicit instantiation and using type parameters to represent associated types. First, verbose calls to generic algorithms are 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 nested type arguments, 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. Second, in addition to being dependent on the function's argument types, call sites of generic algorithms become dependent on all associated types of the argument types. For example, the ***bfs*** function does not take an edge as a parameter, but since it is an associated type of the graph type, it appears as a type parameter to that function. Thus, changing just the associated edge type of the ***ADJACENCY_LIST*** data structure would require revisiting all calls to ***bfs*** with ***ADJACENCY_LIST*** as a type parameter. Section 12.3 gives full details of this problem.

Eiffel does not support constraining a type parameter with multiple classes. To work around this, we define classes that represent combinations of concepts: instead of the two classes ***VERTEX_LIST_GRAPH*** and ***INCIDENCE_GRAPH***, the graph type ***G*** in Figure 6 is required to derive from just one class. This class, ***VERTEX_LIST_AND_INCIDENCE_GRAPH***, inherits from both of the above classes, and adds no new method requirements. The requirements of generic algorithms determine which combinations of concepts need such classes. Adding a new algorithm that requires a previously unused combination of concepts necessitates the creation of a new class for this combination. This problem surfaces with OCaml as well (see Section 6.1), but is more serious in Eiffel: adding a new combination class may require non-local modifications elsewhere in the library. The new class may need to be added to the base classes of concrete graph classes, or to previously defined classes representing combinations of concepts. For instance, a graph type that directly inherits from ***VERTEX_LIST_GRAPH*** and ***INCIDENCE_GRAPH*** is not usable as an algorithm's input type that is constrained with ***VERTEX_LIST_AND_INCIDENCE_GRAPH***. In OCaml, this phenomenon does not occur, as conformance is determined by structural properties, which do not change in this scenario. Multiple constraints is an open issue in Eiffel standardization and a likely future addition to the language (Meyer, 2002; ECMA, 2005).

Eiffel has been criticized for type-safety problems (Bruce, 1996; Cook, 1989). In particular, type conformance checking is based on its subclass relation, which allows instance variables and parameters of routines to change covariantly. Eiffel applies covariance to generic parameters as well, which makes the type-safety problems a concern for generic programming. Under these rules the subclass relation does not imply substitutability, and additional measures must be taken to guarantee type-safety. Suggested approaches include a link-time *system validity check* (Meyer, 1992) that requires a whole-program analysis, and a ban of so-called *polymorphic catcalls* (Meyer, 1995). A recent proposal (Howard *et al*., 2003) requires programmers to add explicit handler functions for each potentially type-unsafe program point. The current draft specification of the Eiffel standard (ECMA, 2005) proposes two measures: a keyword ***frozen*** to signify that no covariant change is allowed on a particular type parameter, and requiring the type checker to force programmers to add *object tests* (checked dynamic casts) where covariance may compromise type safety. To our knowledge, no publicly available compiler implements any of these approaches, and they are thus not considered in our evaluation.

According to the current 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, under 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 more often. 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 ***WHITEN_VERTEX*** algorithm above sets the state of a given vertex to ***WHITE***. The 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 ***put(key: V; val: INTEGER)*** to be the signature of the ***put*** method. 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]***, ***D_VERTEX*** inherits from ***B_VERTEX***, and ***D_INT*** from ***INTEGER***. Now ***MY_MAP[D_VERTEX, D_INT]*** conforms to the class ***READWRITE_MAP[B_VERTEX, INTEGER]***, making the following instantiation seemingly valid:

```
WHITEN_VERTEX[B_VERTEX, MY_MAP[D_VERTEX, D_INT]]
```

In this instantiation, the ***put*** method is called with arguments of types ***B_VERTEX*** and ***INTEGER***. This is an obvious error, as the signature of the method is ***put(k: D_VERTEX; val: D_INT)***. 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 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. Suppose, however, 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 any additional capabilities of the original type.

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. We resorted to covariance only to reduce difficulties arising from the lack of implicit instantiation and support for associated types. In particular, 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. The restrictions introduced to guarantee type safety lead to the compiler rejecting code for reasons that are not easy for a programmer to discern.

```
public interface VertexListGraph<Vertex,
      VertexIterator extends Collection<Vertex>,
      VerticesSizeType extends Integer> {
   VertexIterator vertices();
   VerticesSizeType num_vertices();
}

public interface IncidenceGraph<Vertex, Edge,
      OutEdgeIterator extends Collection<Edge>,
      DegreeSizeType extends Integer> {
   OutEdgeIterator out_edges(Vertex v);
   DegreeSizeType out_degree(Vertex v);
}

public class adjacency_list
   implements VertexListGraph<Integer, Collection<Integer>, Integer>,
              IncidenceGraph<Integer, adj_list_edge<Integer>,
                 Collection<adj_list_edge<Integer>>, Integer>
{ ... }
```

Fig. 8. Two interfaces representing graph concepts in Java, and an adjacency list data structure that models these concepts.

9 Graph library in Java

Java 5 includes generics with type parameters for classes, interfaces, and methods (Gosling *et al.*, 2005). We used version 5.0 of the JDK to compile and test our implementation of the graph library. Java generics follow a similar design to those in Eiffel, but with several differences that proved to be significant. In Java generic methods can be implicitly instantiated and type parameters can be constrained to be subtypes of multiple other types. By default, all type parameters must be subtypes of ***Object***; all user-defined types satisfy this property, but primitive types such as ***int*** and ***double*** do not. A unique feature of Java is so-called *wildcards*, which can be characterized as unnamed type parameters (Torgersen *et al.*, 2004). In addition to subtype constraints, wildcards can be constrained from below, that is, required to be supertypes of particular types.

9.1 Implementation

In Java, we represent concepts by interfaces. A type declares that it models a concept by implementing the corresponding interface. Figure 8 shows the Java representations of two 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. Although not shown in this figure, concept refinement is represented by inheritance between interfaces using the ***extends*** keyword.

As in the Eiffel implementation, associated types are expressed as type parameters to the interface representing the concept. For example, the ***IncidenceGraph*** interface in Figure 8 has three associated types: ***OutEdgeIterator***, ***Vertex***, and ***Edge***. The adjacency list graph type demonstrates connecting concrete associated types to the

```
public class breadth_first_search {
  public static <
    GraphT extends VertexListGraph<Vertex, VertexIterator, ?> &
                   IncidenceGraph<Vertex, Edge, OutEdgeIterator, ?>,
    Vertex,
    Edge extends GraphEdge<Vertex>,
    VertexIterator extends Collection<Vertex>,
    OutEdgeIterator extends Collection<Edge>,
    ColorMap extends ReadWritePropertyMap<Vertex, ColorValue>,
    Visitor extends BFSVisitor<GraphT, Vertex, Edge>>
  void go(GraphT g, Vertex s, ColorMap c, Visitor vis);
}
```

Fig. 9. Breadth-first search interface using Java.

type parameters of the ***IncidenceGraph*** interface: the vertex type is ***Integer***, the edge type is ***adj_list_edge<Integer>***, and so on.

Generic algorithms are most straightforwardly expressed as free-standing functions. In a language not supporting them, they can be emulated by static methods, either as parameterized methods in non-parameterized classes or non-parameterized methods in parameterized classes (as in Eiffel). Parameterized methods in Java can be implicitly instantiated, and so we use them to implement generic algorithms. Following the convention we chose with Eiffel, we name the static method ***go***.

Figure 9 shows the interface for the breadth-first search algorithm. This algorithm takes seven type parameters and four method parameters. Four of the type parameters are types of method parameters; the other three (plus ***Vertex***) are associated types of the graph type. Since Java supports implicit instantiation, calling a generic method is no different from calling a non-generic method. For example, the breadth-first search algorithm is invoked as follows:

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

9.2 Evaluation of Java

Overall, Java provides enough support for generic programming to allow a type-safe implementation of the generic graph library. Interfaces provide a mechanism to represent concepts within the language. Type parameter constraints can be expressed directly and are enforced by the compiler, and multiple constraints can be applied to the same type parameter. The arguments to generic algorithms and the bodies of those algorithms are type checked separately against the concept requirements, leading to good separation between the implementations of data structures used as arguments to generic algorithms and the implementations of those algorithms, and early detection of type errors. This also enables separate compilation of generic components and their uses, which allows for faster compilation. Type arguments to generic functions are automatically deduced from calls to those functions, greatly reducing verbosity and increasing modularity.

Two problems related to subtyping-constrained polymorphism, and some additional inconveniences, make generic programming in Java both restricted and cumbersome. The first 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 12.4 explains how this problem affects the task of composing libraries.

The second problem relates to representing associated types. Java classes and interfaces can only encapsulate methods and member variables, not types. Hence, as in Eiffel, each concept's associated types are represented as separate type parameters. Referring to a concept in a generic algorithm requires repeating all its type arguments. Representing associated types as type parameters results in unnecessarily lengthy code and repetition of the type constraints. Wildcard types can alleviate this to some extent: a ***?*** symbol can be used in place of a type argument. Any constraints on that type argument are assumed and need not be given. If a type argument needs to be referred to, for example, to express that two associated types of different concepts must be the same, then it cannot be replaced with a wildcard. This proved to be the common case in the graph library implementation; wildcards did reduce the verbosity to an extent but often they were not applicable. Sections 12.1 and 12.2 discuss in more detail the problems caused by representing associated types as type parameters.

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 object of type ***MyClass*** is:

```
MyClass x = new MyClass(...);
```

In this example, the name of the type is short. However, this is often not the case in generic libraries. Many types used in the graph library implementation span several lines; see Section 12.5 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++. Wildcards cannot be used for this purpose, as a wildcard defines a separate type, not just an automatically inferred type name. In other words, an expression like ***A<?> a = new A<B>(...);*** cannot be used in many cases, as ***A<?>*** defines a different, and more restricted, type than ***A<B>***. In particular, once the variable ***a*** is given the type ***A<?>***, all information that the type argument to ***A*** is ***B*** is lost to the compiler.

One of the design goals of Java generics was backward compatibility with existing Java implementations. For this reason, the language is implemented using type erasure. Certain type constraints cannot be expressed with a type-erasure-based language, such as constraining one type variable to be a subtype of two different parameterizations of a single generic interface. These restrictions cannot be resolved in Java without sacrificing backward compatibility (Bracha *et al.*, 1998). None of the restrictions of type erasure were encountered in the graph library implementation.

10 Graph library in C# generics

C# generics extend the C# programming language with parameterized classes, interfaces, and methods (Kennedy & Syme, 2001; Microsoft Corporation, 2005).

```
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 class adjacency_list
  : VertexListGraph<int, IEnumerable<int>>,
    IncidenceGraph<int, adj_list_edge<int>,
                   IEnumerable<adj_list_edge<int>>
{ ... }
```

Fig. 10. C# representations of two graph concepts and a type that models the concepts.

Apart from minor syntactic differences, generic class and method definitions are similar to those in Java. Full separate type checking and compilation are supported. Generics are a planned feature for version 2.0 of the .NET Framework, and a beta release of that framework was used for compiling the graph library.

10.1 Implementation

Generic programming structures are implemented in C# generics using many of the same techniques as are used in Java. Concepts are represented as interfaces, and modeling a concept is represented as implementing the corresponding interface. Refinement is expressed by one interface inheriting from another. Generic algorithms are implemented by parameterized static methods, and concept constraints on the parameters to those algorithms are represented by interface-based constraints. Also as in other object-oriented languages, associated types are realized as extra type parameters to interfaces and algorithms.

Figure 10 shows the interfaces representing the Vertex List Graph and Incidence Graph concepts, as well as a graph data structure modeling these concepts.

The interface to the breadth-first search algorithm is shown in Figure 11. The example invocation of this algorithm in Figure 12 illustrates the most significant difference between Java and C# generics: the current language specification for C# does not allow the inference of type arguments to a generic function from the constraints of that function. For example, the following code:

```
public interface I<T> {}
public class C: I<int> {}
public static class Algorithm {
  public static void go<T, U>(U b) where U: I<T> {}
}
```

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

Fig. 11. Breadth-first search interface in 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>>,
  simple_property_map<int, ColorValue>,
  my_bfs_visitor<adjacency_list, int, adj_list_edge<int>>>
(graph, src_vertex, color_map, visitor);
```

Fig. 12. Call to breadth-first search algorithm in C#.

```
public static class MainClass {
  public static void Main(string[] args) {
    C x = new C();
    Algorithm.go(x);
  }
}
```

does not work in the current version of C#, because the type argument ***T*** to ***Algorithm.go()*** cannot be inferred, as ***T*** only occurs in the constraints of the ***go()*** method, not in its parameter list. This problem can be avoided in either of two ways: by using explicit instantiation in the cases where the type inference problem occurs, which we chose for the C# implementation of the graph library; or by compromising the design of the generic library to allow implicit instantiation by replacing type parameters with their bounds, as we did in the Cecil implementation of the graph library (Cecil has a similar limitation). The problems with both of these workarounds are explained in Section 12.3. Since Java is capable of inferring type arguments from constraints, it does not have this problem.

10.2 Evaluation of C# generics

C# generics avoid some of the inconveniences of Java; in particular, primitive types can be used directly as arguments to generic components. Otherwise, its suitability for generic programming is almost the same as that of Java. Like Java, C# does not support a representation for one concept simultaneously constraining multiple independent types; both languages support multiple constraints on the same type parameter. Associated types are represented as extra type parameters to interfaces and generic methods, as in Java. The problems related to this are the same in both languages, including added verbosity and the inability to encapsulate constraints on associated types as part of a concept. C#, like Java, lacks a mechanism for type aliases, although there is a syntax for defining shorthand names for concrete types outside of a generic class or method. Because of this restriction, a type containing type variables cannot be abbreviated in this way. Unlike Java, C# supports retroactive modeling, though in a limited form, through the use of partial classes. A class must be defined in a particular way to be extensible through that mechanism, however, and all base interfaces must be found at compile-time; this means, for example, that classes imported from a library in binary form cannot be made to retroactively model concepts (Microsoft Corporation, 2005, ch. 23). C#, similar to the other object-oriented languages studied, supports separate compilation for generics.

A severe drawback of the .NET Framework 2.0 beta implementation of generics in C# is its support for implicit instantiation, which we found insufficient for convenient implementation of the graph library. The need to explicitly specify all type arguments to generic methods leads to unnecessary type duplication and a loss of modularity, as discussed in Sections 8.2 and 12.3.

11 Graph library in Cecil

Cecil is a research object-oriented language with support for subtype-constrained polymorphism on both objects and methods (Litvinov, 1998; Chambers & the Cecil Group, 2002). Cecil provides a static type system on top of a dynamically typed core language. The static type system only generates warnings and does not reject any code; error-producing type checks are all done at run-time and are based on the dynamic types of objects. Cecil is prototype-based and thus does not have classes: objects serve the same purposes as classes do in other languages. In particular, objects contain fields and methods, and inherit behavior from other objects. An *abstract object* is the analogue of an abstract class or interface in other object-oriented languages; a *template object* corresponds to a generic class.

Generics in Cecil, compared to Java, Eiffel, or C#, add several features: multimethods, retroactive abstraction, and more powerful inference of type parameter constraints. Multimethods bind method definitions to a combination of several objects, not just to a single object, which is the case in mainstream object-oriented languages. Retroactive abstraction allows subtype relationships, and other object features, to be added to a type after it has been defined; this contrasts with several popular object-oriented languages, in which the superclasses, fields, and methods of a class are set at class definition time and cannot be changed elsewhere in the program. Finally, Cecil can infer constraints on type parameters, based on their uses in other constraint expressions. Our evaluation is based on version 3.2a of the Cecil interpreter, prepared by Craig Chambers.

```
abstract object vertex_list_graph[‘Vertex,
                                  ‘VertexIterator <= collection[Vertex]];
  signature vertices(:vertex_list_graph[‘Vertex, ‘VertexIterator]):
                                                        VertexIterator;
  signature num_vertices(:vertex_list_graph[‘Vertex, ‘VertexIterator]): int;

abstract object incidence_graph[‘Vertex, ‘Edge <= bgl_graph_edge[Vertex],
                                ‘OutEdgeIterator <= collection[Edge]];
  signature out_edges(:incidence_graph[‘Vertex, ‘Edge, ‘OutEdgeIterator],
                      :Vertex): OutEdgeIterator;
  signature out_degree(:incidence_graph[‘Vertex, ‘Edge, ‘OutEdgeIterator],
                       :Vertex): int;
```

Fig. 13. Two abstract objects representing graph concepts in Cecil.

```
template object adjacency_list
  isa vertex_list_graph[int, array[int]],
     incidence_graph[int, adj_list_edge[int], array[adj_list_edge[int]]],
     edge_list_graph[int, adj_list_edge[int], array[adj_list_edge[int]]];

  field vertices_(@:adjacency_list): array[int] := object isa array[int];
  method vertices(g@:adjacency_list): array[int] {g.vertices_}
  method num_vertices(g@:adjacency_list): int {g.vertices_.length}
  -- Other field and method definitions for adjacency_list
```

Fig. 14. Partial listing of the adjacency list data structure that models several graph concepts.

11.1 Implementation

Concepts in Cecil are represented as abstract objects. As an example, the first two lines in Figure 13 define the abstract object for the Vertex List Graph concept. The ***‘Vertex*** syntax, a backquote preceding an identifier, is a concise syntax for declaring a type parameter. Any later occurrences of the same identifier are written without the backquote and refer to the same type parameter. Subtype constraints follow a type parameter declaration after a <= symbol. A feature peculiar to Cecil is that the method requirements, denoted using the ***signature*** keyword, are completely separate from the definitions of the abstract objects. The two ***signature*** declarations in Figure 13 following the declaration of ***vertex_list_graph*** demonstrate this. Functions with these signatures must be defined for all objects inheriting from ***vertex_list_graph***. The separation of the object declaration and the method signatures is a natural consequence of Cecil's support for multimethods: no parameter of a method is special (such as the ***this*** parameter in Java or C#). A concept in Cecil thus consists of a declaration of an abstract object, and a set of signature definitions on that object.

In Cecil, types are declared to model concepts by establishing the subtyping relationship with the ***isa*** keyword. The declaration of the ***adjacency_list*** template object and the definitions of some of its fields and methods are shown in Figure 14. Similar to the signatures of abstract objects, field and method definitions are separate from the template object declaration.

```
method breadth_first_search(
  g: ‘Graph <= incidence_graph[‘Vertex, ‘Edge, ‘OutEdgeIterator]
                & vertex_list_graph[Vertex, ‘VertexIterator],
  s: Vertex,
  color: m_table_like[Vertex, color_value],
  vis: bfs_visitor[Graph, Vertex, Edge]): void { ... }
```

Fig. 15. Breadth-first search interface using Cecil.

Cecil’s support for multimethods allows generic algorithms to be defined as methods that do not belong to any particular object. Figure 15 shows the signature of the breadth-first search algorithm. Cecil supports multiple constraints on a single type parameter, demonstrated by the ***Graph*** type parameter: the <= symbol is followed by the list of constraints separated by **&** symbols. The ***:*** symbol is used to specify the static type of a method parameter or return type; the @ symbol restricts the parameter at run-time to inherit from a particular object. We follow Cecil conventions in the use of ***:*** and @.

Cecil supports implicit instantiation; thus, the type arguments to generic algorithms do not need to be passed explicitly, and a call to a generic method is no different from a call to a non-generic method. Cecil’s argument deduction capabilities are, however, less powerful than those of Java. In particular, Cecil cannot infer type arguments from a generic method’s constraints; C# generics also share this restriction. The visitor type in the breadth-first search algorithm implementation is not given a separate type parameter to work around this limitation. We discuss this issue in more detail in Section 12.3.

As in other object-oriented languages, associated types of a concept are extra type parameters to the abstract object representing the concept. For example, the ***incidence_graph*** abstract object in Figure 13 has three associated types, and constraints on them. These same associated types appear in the ***breadth_first_search*** algorithm in Figure 15. Unlike Java and C#, the constraints on associated types do not need to be repeated; they are inferred from the constraints placed on the ***Graph*** type parameter. For example, the constraint that ***OutEdgeIterator*** be a collection does not need to be specified in ***breadth_first_search*** as it is already specified in the ***incidence_graph*** concept, shown in Figure 13.

In Cecil, retroactive modeling is implemented with retroactive abstraction: the subtyping relation between two types can be established with a declaration separate from the definitions of the types. Moreover, subtyping declarations can be generic, and apply only to instantiations where type parameters satisfy predefined constraints. The following code demonstrates these capabilities. The ***bgl_graph_edge*** object represents the Graph Edge concept we used to describe the requirements for edge types. The ***adj_list_edge*** object is a model of that concept.

```
abstract object bgl_graph_edge[‘Vertex];
  signature source(:bgl_graph_edge[‘Vertex]): Vertex;
  signature target(:bgl_graph_edge[‘Vertex]): Vertex;

template object adj_list_edge[‘Vertex] isa bgl_graph_edge[Vertex];
  field source_(@:adj_list_edge[‘Vertex]): Vertex;
  field target_(@:adj_list_edge[‘Vertex]): Vertex;
  method source(e@:adj_list_edge[‘Vertex]): Vertex {e.source_}
  method target(e@:adj_list_edge[‘Vertex]): Vertex {e.target_}
```

Equality comparison between ***adj_list_edge***s can be defined based on equality of vertices. The following code makes ***adj_list_edge*** retroactively a model of Comparable, in all cases where its type parameter ***Vertex*** models Comparable:

```
extend adj_list_edge[‘Vertex <= comparable[Vertex]]
    isa comparable[adj_list_edge[Vertex]];
  method =(e1@:adj_list_edge[‘Vertex <= comparable[Vertex]],
            e2@:adj_list_edge[Vertex]): bool {
    (e1.source_ = e2.source_) & (e1.target_ = e2.target_)
  }
```

Note that this particular subtyping relation must be defined retroactively because not all instances of ***adj_list_edge*** implement the functionality of Comparable.

11.2 Evaluation of Cecil

Cecil is an object-oriented language with some advanced features not found in mainstream object-oriented languages. Of these features, inference of type parameter constraints most significantly affects generic programming. This manifests as Cecil’s improved support for associated types, as compared to Java and C#; constraints placed on associated types in concept definitions need not be repeated when the concepts are used to constrain type parameters. Associated types must, however, still be given as extra type parameters in each generic algorithm, whether or not those types are used. As a less notable effect, Cecil’s backquote syntax for declaring type parameters makes generic definitions shorter. Type parameter constraints and the function parameter list, however, become interleaved, which may not have an entirely positive effect on code readability. Finally, Cecil provides retroactive abstraction: the subtype relation is not fixed when particular classes or objects are defined; hence, Cecil supports retroactive modeling and does not suffer from the problems described in Section 12.4.

The current implementation of Cecil cannot infer type arguments from constraints; Cecil shares this problem with C#. To preserve implicit instantiation we had to round some type parameters to their constraints; Section 12.3 discusses this issue in more detail.

Cecil has an undocumented polymorphic type aliasing mechanism (based on a personal communication from Craig Chambers), but with similar restrictions to the ***using*** declaration in C#, and so is not very useful for generic libraries; thus, some type name duplication occurs in generic algorithms. Cecil’s syntax for variable declaration, however, does not require duplication of type names, which makes the problem less pronounced than in the other object-oriented languages we studied.

Defining methods separately from objects is convenient in many cases, but also necessitates declaring the type parameters for each method of the same object. This is occasionally repetitive. For example, in the implementation of Dijkstra’s shortest

```
method initialize_vertex(
    @: dijkstra_visitor['Graph, 'Vertex, 'Distance, 'Edge, 'QueueType,
                        'WeightMap, 'PredecessorMap, 'DistanceMap,
                        'DistanceCombine, 'DistanceCompare],
    u: Vertex, g: Graph): void {}
method discover_vertex(
    @: dijkstra_visitor['Graph, 'Vertex, 'Distance, 'Edge, 'QueueType,
                        'WeightMap, 'PredecessorMap, 'DistanceMap,
                        'DistanceCombine, 'DistanceCompare],
    u: Vertex, g: Graph): void {}
```

Fig. 16. Two of the eight methods of the ***dijkstra_visitor*** object used in the implementation of Dijkstra's shortest path algorithm.

paths algorithm (see Figure 16) the type of the visitor object with all ten of its type parameters is repeated for each of the eight methods of the visitor.

Finally, to support the greater flexibility, Cecil requires a whole-program type check to prevent ambiguous and undefined method calls (Chambers & the Cecil Group, 2002, § 3.3).

12 Discussion

The following sections describe six specific issues brought to light in the course of this study. The first regards limitations in how subtyping is used to constrain type parameters in mainstream object-oriented languages. The second involves repercussions of the way associated types are accessed in the surveyed languages. The third emphasizes how explicit instantiation (or weaknesses in implicit instantiation) combined with insufficient support for representing associated types can make generic function calls dependent on the implementation details of their argument types. The fourth concerns the effect of the type constraint mechanism on the evolution of software systems, especially when new algorithms and concepts are created. The fifth shows how the lack of a mechanism for type aliasing can force unnecessary verbosity, especially when generic components must be explicitly instantiated. Finally, the sixth discusses some aspects of language syntax that can discourage the development of generic libraries.

12.1 Encapsulating type constraints in concepts

In all the purely object-oriented languages under study, we implemented generic programming using analogous language facilities. Concepts are approximated in Java and C# by interfaces, in Eiffel by deferred classes, and in Cecil by abstract objects (in the following we use the term "interface" to refer to all of these). The modeling relation between a type and a concept is approximated by the subtype relation between a type and an interface, and the refinement relation between two concepts by interface extension.

Concepts commonly group constraints concerning multiple types. For example, the Vertex List Graph concept places constraints on a graph type as well as its associated vertex and vertex iterator types. In particular, the Vertex List Graph concept requires that the vertex iterator type models the Collection concept. In the following, we attempt to express the Vertex List Graph concept using a parameterized interface with extra constraints on the type parameters. However, this attempt ultimately fails. Consider the following Java implementation of the Vertex List Graph concept:

```
interface VertexListGraph<Vertex,
      VertexIterator extends Collection<Vertex>,
      VerticesSizeType extends Integer> {
   VertexIterator vertices();
   VerticesSizeType num_vertices();
}
```

The ***VertexIterator*** parameter is constrained to require that it models the Collection concept. Difficulties arise when ***VertexListGraph*** is later used to constrain an algorithm's graph type parameter. For example, in the following sketch of a ***graph_algorithm*** function we constrain ***G*** to extend ***VertexListGraph***.

```
public static class graph_algorithm {
  public static <
    G extends VertexListGraph<Vertex, VertexIterator, ...>,
    Vertex,
    VertexIterator>
  void go(G g, Vertex s, VertexIterator i);
}
```

The above code fails to type check: the instantiation of ***VertexListGraph*** with the type argument ***VertexIterator*** fails because ***VertexIterator*** does not implement ***Collection<Vertex>***. A constraint on an interface's type parameter is a prerequisite to instantiating the interface; it is not part of the interface itself. Thus the constraint on the ***VertexIterator*** parameter resulted in the opposite behavior to what we intended: the ***VertexListGraph*** constraint on the method ***go*** does not allow us to assume that ***VertexIterator*** implements ***Collection*** but instead requires us to prove that ***VertexIterator*** implements ***Collection***.

The above code can be made to type check by adding a constraint on the ***VertexIterator*** parameter of the ***go*** method:

```
public static class graph_algorithm {
  public static <
    G extends VertexListGraph<Vertex, VertexIterator, ...>,
    Vertex,
    VertexIterator extends Collection<Vertex>, ...>
  void go(G g, Vertex s, VertexIterator i);
}
```

This need to repeat the constraint implies that the Java interfaces (and their equivalents in C# and Eiffel) cannot organize and group constraints on multiple types. Instead the constituent constraints of such concepts must be repeated for every generic algorithm.

Approximating concept refinement with interface extension exhibits similar effects. For example, the following Java declaration is not legal:

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

The ***OutEdgeIter*** type must be declared to extend ***Collection<Edge>*** before instantiating ***IncidenceGraph***. Here is the corrected version:

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

Java, C#, and Eiffel all exhibit the above effects. In contrast, Cecil does not require redundant constraint declarations analogous to those illustrated above. Our recent work (Järvi *et al*., 2005) suggests changes to the type systems of Java and C# to obviate redundant constraints.

12.2 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. In C++, the type expression ***graph_traits<G>::vertex*** exemplifies how the ***graph_traits*** traits class provides a mapping from a graph type ***G*** to its vertex type. In SML, such a mapping is accomplished by nesting types within structures. The other languages we evaluated do not currently provide a direct mechanism for expressing functions over types that can dispatch on their inputs, as is needed for associated types; work towards such features exists for Haskell (Chakravarty *et al*., 2005a) and in the context of C# and Java (Järvi *et al*., 2005).

One way to represent associated types is to represent them by adding a parameter for each associated type to the interface (Java, C#, OCaml, Cecil, and Eiffel) or type class (Haskell) representing the concept. Then, these associated types can be accessed in generic functions by similarly declaring new type parameters, and using them as type arguments to the interfaces. This approach results in a significant increase in the verbosity of both the type parameterization and constraints of generic components. For example, the Java version of ***breadth_first_search*** shown in Figure 9 includes seven type parameters, four of which are associated types of the graph. In contrast, the C++ version of ***breadth_first_search*** has only three type parameters, as the associated types of the graph type can be accessed directly without extra type parameters. The verbosity introduced by this approach compounds the problem of repeated constraints discussed in Section 12.1.

12.3 Implicit instantiation

In languages that cannot encapsulate associated types, and do not support implicit instantiation (Eiffel) or only support a weak form of implicit instantiation (C#), explicit instantiation results in verbose generic algorithm invocations. The C# call to breadth-first search in Figure 12 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 argument types 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 level of support for implicit instantiation, or type argument deduction, proved significant in our evaluation. C# and Cecil lack the ability to determine the type arguments based on the constraints of a generic function, whereas Java supports this capability. For example, the following C# function cannot be called without explicitly specifying the type argument ***U***:

```
public static class Algorithm {
  public static void go<T, U>(U u) where U: IEnumerable<T> { ... }
}
```

In contrast, C# implicit instantiation can handle the slightly different definition:

```
public static class Algorithm {
  public static void go<T>(IEnumerable<T> u) { ... }
}
```

Such "rounding parameter types to their bounds" may lead to problems in generic algorithms because the exact static types of the arguments are not available. The exact type may be needed in the return type of the function, types of other parameters, or in their constraints. For example, the C# version of the Johnson's all-pairs shortest paths algorithm includes the following type parameters and their constraints:

```
DistanceMatrixElement: ReadWritePropertyMap<Vertex, Distance>
DistanceMatrix: ReadablePropertyMap<Vertex, DistanceMatrixElement>
```

Assume the type arguments in a call are ***DME*** and ***DE*** satisfying the constraints:

```
DME : ReadWritePropertyMap<int, double>
DM : ReadablePropertyMap<int, DME>
```

If the ***DistanceMatrixElement*** type parameter is replaced with its bound, namely ***ReadWritePropertyMap<Vertex, Distance>***, ***DM*** would need to be a subtype of

```
ReadablePropertyMap<int, ReadWritePropertyMap<int, double>>
```

for the call to type check. Typically, this would not be the case, but rather, ***DM*** would only be a subtype of ***ReadablePropertyMap<int, P>***, where ***P*** is a subclass of ***ReadWritePropertyMap<int, double>***. Since ***P*** is not exactly the type ***ReadWritePropertyMap<int, double>***, the required subtyping relation does not hold.

Note that argument deduction in C# succeeds only if all type arguments can be deduced, otherwise every type argument must be explicitly specified. Also, rounding the type of a function argument to its bound is not a transparent operation, but instead can require changes to user code at the call sites of the modified function. Finally, in languages where different instances of generic functions or methods are compiled to separate functions in the generated executable, using the exact type of a function argument, rather than a base class of that type, may result in a performance gain.

We used two different approaches to handle the weak support for implicit instantiation in the implementations of the graph library: the C# version uses full type parameterization, and thus cannot use implicit instantiation for calls to most algorithms; the Cecil version rounds some type parameters to their bounds (as was done with the example above), removing exact type information but enabling implicit instantiation more often.

Explicitly specifying type arguments that represent associated types also leads to the introduction of unnecessary implementation dependencies. As explained above, four of the type arguments in the call to the breadth-first search algorithm in Figure 12 are associated types of the first type argument, the graph type. These types may represent internal implementation details of the graph type. At each call to the algorithm, however, these types must be explicitly specified. Consequently, changes that should be just implementation details in a data structure require changes to user code at the call sites of generic algorithms that pass that data structure as an argument. Consider the call to the breadth-first search algorithm in Eiffel that was shown in Section 8.2:

```
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 6 uses ***INTEGER*** as the vertex type and ***BASIC_EDGE[INTEGER]*** as the edge type. Even though the breadth-first search algorithm has no edge object as a parameter, the associated edge type must still be specified. Thus, 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]
```

Similarly, all calls which pass an ***ADJACENCY_LIST*** to a graph algorithm which uses a concept with an associated edge type must be updated.

12.4 Establishing the modeling relation

The type arguments to a generic algorithm must model the concepts that constrain the algorithm's type parameters. We use several different techniques to establish modeling relationships in the languages under study. Java, C#, Eiffel, and Cecil use subtyping at the point of class definition; Cecil also allows subtype relationships to be added outside of class definitions. C# supports similar functionality in a limited form: classes have to be declared *partial* for this purpose, and their retroactive extension must occur in the same compilation unit as their definition. Haskell requires an explicit ***instance*** declaration independent of data type definition. Modeling relationships in SML (and OCaml) are implicitly checked for structural conformance at generic function call sites. Any structure that meets the signature (or class type) 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, Cecil, and Haskell rely on named conformance, where an explicit declaration links a concrete type to the concepts it models. Haskell differs from the others in that conformance declarations are usually separate from data structure definitions. Cecil supports named conformance both as part of a type definition and separately. Modeling in SML and OCaml relies on structural conformance. The names of concepts are irrelevant; only the established requirements matter. The modeling mechanisms in SML, OCaml, Haskell, and Cecil worked well for implementing the graph library. 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 (Magnusson, 1991) is a ***rectangle*** class with ***move*** and ***draw*** methods, and a ***cowboy*** class with ***move***, ***draw***, and ***shoot*** methods. With structural conformance, a ***cowboy*** could be used where a ***rectangle*** is expected, possibly resulting in troublesome runtime errors. In our experience, accidental conformance is not a significant source of programming errors.

In languages where modeling is established by named conformance only 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 17 shows in C# an example of the retroactive modeling problem manifest in three distinct libraries. Library ***A*** defines a graph concept Vertex List Graph, as well as a graph data structure ***adjacency_list*** that models that concept. 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 and Cecil solve this by allowing modeling and refinement relationships to be established after the related types and concepts are defined. Languages such as SML and C++ do not encounter this problem, as they use structural conformance to constrain type parameters.

Retroactive subtyping can be addressed by providing a language mechanism external to the class definition that establishes a subtyping relation, either automatically (Baumgartner *et al.*, 2002; Läufer *et al.*, 2000) or by hand using the Adapter pattern (Gamma *et al.*, 1995); Cecil already contains such a mechanism, and it was used to implement the graph library. For example, to indicate that the ***adj_list_edge***

```
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();
  }
  public static 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, ...);
}
```

Fig. 17. An example showing the need for retroactive modeling.

class (used for edges in the adjacency list graph implementation) is a subtype of ***comparable*** (Equality Comparable), the following declaration suffices:

```
extend adj_list_edge['Vertex <= comparable[Vertex]] isa
  comparable[adj_list_edge[Vertex]];
```

After this, the = method must also be defined to implement the ***comparable*** functionality. This process of retroactive subtyping is analogous to how the Haskell ***instance*** declaration establishes a modeling relation. Other solutions to retroactive modeling include aspect oriented programming systems (Kiczales *et al.*, 1997), such as AspectJ, that 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* (Kiczales *et al.*, 2001). The *External Polymorphism* design pattern is an attempt to address this integration problem without changes to existing object-oriented languages (Cleeland *et al.*, 1997).

```
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);
```

Fig. 18. Lack of type aliases leads to unnecessarily lengthy code.

12.5 Type aliases

Type aliases are 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 C# the type of the visitor object used in Dijkstra's algorithm is:

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

In this example, some type arguments are instantiations of other parameterized components; this is not uncommon in generic code. The resulting type name is unwieldy, and code that must use it is likely to be cluttered and hard to read. Furthermore, a long type name may appear many times in a piece of code. This five-line-long type appears three times in 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 18 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 21 lines. There is no mechanism for type aliases in Java, C#, or Eiffel (the ***using*** directive in C# and the ***type synonym*** directive in Cecil provide a limited form of type aliasing, but they cannot be used inside a method, and thus cannot reference the method's type parameters). Type aliasing is a simple but crucial feature for managing the long type expressions that are commonly encountered in generic programming.

12.6 Concise syntax

Our evaluation criteria systematically break down key properties for generic programming across languages. Some language-specific syntactic properties, which seem to impact the process of developing and using generic libraries, are not necessarily sufficiently covered within such a categorization. This section discusses some smaller syntactic issues that can discourage the development of generic libraries.

Standard ML's module system is a powerful mechanisms for combining program components, but it requires substantial syntactic overhead. When combining large-scale components like parameterized modules that contain many functions, values, and types, the syntax for signatures, structures, and functors is minimal compared to the contents of a module. When implementing a single generic algorithm as a functor, however, the amount of text required to define the algorithm, state its requirements, and later apply the functor to a set of valid structures is substantial even compared to other languages that require explicit instantiation. Any increase in syntactic overhead to make functions generic decreases their utility and appeal.

Proposed future extensions to C++ (Stroustrup & Dos Reis, 2005; Siek *et al.*, 2005) with explicit constraints on template parameters will make generic definitions more verbose. Moreover, C++ compilers often display each error message with the entire template instantiation stack, which can be extremely verbose. Due to the lack of constraints, the entire stack is necessary for debugging erroneous template instantiations.

Java has opted for relatively long keywords (***extends***, ***implements***) to express subclassing and interface extension; C# is more concise in this respect but uses ***where*** as the separator between constraints on different type parameters. Cecil's backquote syntax provides an interesting and concise alternative for declaring type variables, and constraints on them, at the points of the first uses of type parameters.

Java and C# require wrapping generic algorithms into classes as static methods. In Eiffel, such wrapping is an even more notable source of syntactic clumsiness: with no support for static methods, generic algorithms must be implemented as normal class methods, meaning that a dummy object must be created in order to invoke the algorithm. Cecil and C++ allow free-standing functions, avoiding these annoyances.

13 Conclusion: Beyond `fold` and `List<T>`

In this study we investigated the support for generic programming across eight programming languages. In particular, we explored how easy or difficult it is to implement a generic graph library in each of the languages, thus going beyond the simple uses of generics for classes such as ***List***<***T***>. For the purposes of this study, the languages fell into three groups: the functional programming languages, the object-oriented languages, and C++.

The support for generic programming in the functional languages was based on parametric polymorphism and proved expressive enough to easily implement a complex generic library, though with some minor issues. The object-oriented languages augmented the parametric polymorphism with subtype bounds. In general, we found it somewhat more difficult to use object-oriented interfaces and subtyping to express the requirements of a generic algorithm. C++ templates represent an altogether different approach to generics that provides a flexible, expressive, and efficient tool for generic programming but is plagued by modularity problems. Unlike the functional and object-oriented languages, C++ template are not based on parametric polymorphism; they are instead more closely related to macros.

We found that the functional languages in our study – Standard ML, Haskell, and OCaml – provide an informative perspective on the design space for generics. These three languages provide both polymorphic functions and data types, which together are sufficient to implement polymorphic algorithms such as ***fold*** that operate on polymorphic data structures. However, these languages go beyond basic parametric polymorphism to provide more powerful mechanisms for organizing abstractions: type classes in Haskell, signatures and functors in SML, and object types in OCaml. As a result, these languages are well suited for generic programming.

In particular, the SML module system supports access to associated types, separate type checking, and static type safety. Haskell and OCaml improve on SML generic programming support with implicit type argument deduction for calls to generic functions. Haskell's type classes are a close match to concepts in generic programming, which made it straightforward to map from the design of the generic graph library to the Haskell implementation. These three languages, however, have their disadvantages. SML lacks functions with constrained genericity aside from functions nested within functors. Haskell and OCaml are quite expressive, but associated types must be represented as extra type parameters to type classes and class types respectively. This formulation forces associated types to be named at every use of a concept. In OCaml, one cannot use type annotations in a function signature to explicitly constrain a polymorphic function's implementation. Nonetheless, these languages demonstrate that type systems based on parametric polymorphism provide a reasonable substrate upon which to build generic libraries.

The C++ template system provides a powerful tool for generic programming. Access to associated types can be implemented through traits classes, and implicit type argument deduction eases the burden for clients of generic functions. Though C++ templates permit the construction of useful generic libraries, its support for generic programming has some limitations, especially in the area of modularity. It lacks separate compilation and separate type checking. Furthermore, argument-dependent lookup makes it difficult to follow which operations a generic function may call and makes it possible for name clashes to occur between separate namespaces. The most notable limitation of C++ is that it does not support explicit expression and enforcement of concepts. However, direct support for concepts are currently under consideration for the next revision of the C++ standard (Siek *et al.*, 2005; Stroustrup

& Dos Reis, 2005). In spite of these flaws, large-scale industrial strength generic libraries are successfully written in C++.

Mainstream object-oriented programming languages have begun to include support for generics. Java 5 now includes generics, and C# generics have been accepted and are expected in a future release of that language. 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. However, in this study we investigated whether generics as they appear in these object-oriented languages can be applied to the broader scope of generic programming, that is, in the construction of libraries of generic algorithms.

In the object-oriented languages studied, constraints on generics are expressed in terms of subtyping. This caused difficulties in dealing with associated types, both in accessing and placing constraints on them. Most of the object-oriented languages failed to provide a mechanism for practical retroactive modeling (with Cecil as the notable exception). All of the object-oriented languages, with the exception of Eiffel, support implicit instantiation. This feature, however, could not be fully exploited in C# and Cecil, due to the weak form of type argument deduction in these languages. Across the board, the object-oriented languages failed to provide a practical mechanism for type aliasing, which caused a serious explosion in the size of type annotations in the code. On the positive side, most of the object-oriented languages provide separate compilation. Overall, implementing a library of generic algorithms in the object-oriented languages was considerably more difficult than in either the functional languages or C++.

By exercising many languages with respect to their support for the generic programming style, we have shown that genericity does not stand alone: closely related language features can have a dramatic impact on the expressiveness of languages. Some authors of this paper have used it as a springboard for the design of new languages that better support generic programming (Siek & Lumsdaine, 2005a; Siek, 2005; Siek & Lumsdaine, 2005b). By highlighting the positive aspects of languages and noting areas worthy of further study, we hope to have opened the doors to continued exploration of this interesting and useful set of language tools.

Acknowledgments

This work was supported by NSF grants EIA-0131354, and ACI-0219884, and by a grant from the Lilly Endowment. The fifth author was supported by a Department of Energy High Performance Computer Science Fellowship. The authors thank Todd Veldhuizen and the anonymous referees for their careful reading of this paper and for their well-considered comments.

References

Austern, M. H. (1998) *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Boston, MA, USA: Addison-Wesley.

Backhouse, R., Jansson, P., Jeuring, J. & Meertens, L. (1999) Generic programming – an introduction. In: *Pages 28–115 of:* S. D. Swierstra, H. Doaitse, P. Rangel and J. M. Oliveira (eds.), *Advanced Functional Programming, Third International School*, Braga, Portugal, revised lectures. Lecture Notes in Computer Science, vol. 1608. Springer-Verlag.

Baumgartner, G., Jansche, M. & Läufer, K. (2002) *Half & Half: Multiple Dispatch and Retroactive Abstraction for Java*. Tech. rept. OSU-CISRC-5/01-TR08. Ohio State University.

Bellman, R. (1958) On a routing problem. *Quart. J. Appl. Math.* **16**(1), 87–90.

Bracha, G., Odersky, M., Stoutamire, D. & Wadler, P. (1998) Making the future safe for the past: adding genericity to the Java programming language. *Pages 183–200 of: OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press.

Bruce, K. B. (1996) *Typing in object-oriented languages: Achieving expressibility and safety*. Tech. rept. Williams College.

Chakravarty, M. M. T., Keller, G. & Peyton Jones, S. (2005a) Associated type synonyms. *ICFP '05: Proceedings of the International Conference on Functional Programming*. ACM Press.

Chakravarty, M. M. T., Keller, G., Peyton Jones, S. & Marlow, S. (2005b) Associated types with class. *Pages 1–13 of: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press.

Chambers, C. & the Cecil Group. (2002) *The Cecil language: Specification and rationale, version 3.1*. University of Washington, Computer Science and Engineering. http://www.cs.washington.edu/research/projects/cecil/.

Cleeland, C., Schmidt, D. C. & Harrison, T. H. (1997) External polymorphism – an object structural pattern for transparently extending C++ concrete data types. In: Martin, R. C., Riehle, D. & Buschmann, F. (eds.), *Pattern Languages of Program Design*. Software Pattern Series, vol. 3. Addison-Wesley.

Cook, W. R. (1989) A proposal for making Eiffel type-safe. *The Comput. J.* **32**(4), 304–311.

Dijkstra, E. W. (1959) A note on two problems in connexion with graphs. *Numerische Mathematik*, **1**, 269–271.

ECMA (2005) *Standard: Eiffel analysis, design and programming language*. ECMA International. Draft 5.00.00-1.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns: Elements of reusable object-oriented software*. Professional Computing Series. Addison-Wesley Longman.

Garcia, R., Järvi, J., Lumsdaine, A., Siek, J. & Willcock, J. (2003) A comparative study of language support for generic programming. *Pages 115–134 of: OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*. ACM Press.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005) *The Java language specification, third edition*. Addison-Wesley Longman.

Graph Library URL (2005) Available at http://www.osl.iu.edu/research/comparing/.

Haack, C. & Wells, J. B. (2003) Type error slicing in implicitly typed higher-order languages. In: *Pages 284–301 of:* Degano, P. (ed.), *Programming languages and systems: 12th European Symposium on Programming, ESOP 2003,* Warsaw, Poland. Lecture Notes in Computer Science, vol. 2618. New York, NY: Springer-Verlag.

Hinze, R. & Jeuring, J. (2003) Generic Haskell: Practice and theory. In: *Pages 1–56 of:* Backhouse, R. & Gibbons, J. (eds.), *Generic programming: Advanced lectures*. Lecture Notes in Computer Science, vol. 2793. Springer-Verlag.

Howard, M., Bezault, Eric, Meyer, Bertrand, Colnet, Dominique, Stapf, Emmanuel, Arnout, K. & Keller, M. (2003) *Type-safe covariance: competent compilers can catch all catcalls.* http://www.inf.ethz.ch/~meyer/.

Järvi, J., Willcock, J. & Lumsdaine, A. (2005) Associated types and constraint propagation for mainstream object-oriented generics. *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications.* To appear.

Jazayeri, M., Loos, R., Musser, D. & Stepanov, A. 1998 (Apr.) Generic Programming. *Report of the Dagstuhl seminar on generic programming.*

Jeuring, J. & Jansson, P. (1996) Polytypic programming. In: *Pages 68–114 of:* Launchbury, J., Meijer, E. & Sheard, T. (eds.), *Advanced functional programming, second international school-tutorial text*. Lecture Notes in Computer Science, vol. 1129. Springer-Verlag.

Jones, M. P. (2000) Type classes with functional dependencies. *Pages 230–244 of: ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems.* Lecture Notes in Computer Science, vol. 1782. Springer-Verlag.

Kennedy, A. & Syme, D. (2001) Design and implementation of generics for the .NET Common Language Runtime. *Pages 1–12 of: PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation.* ACM Press.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997) Aspect-oriented programming. In: *Pages 220–242 of:* Akşit, M. & Matsuoka, S. (eds.), *ECOOP '97 – Object-oriented Programming 11th European Conference*, Jyväskylä, Finland. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001) An overview of AspectJ. In: *Pages 327–353 of:* Knudsen, J. L. (ed.), *ECOOP 2001 – Object-oriented Programming 15th European Conference.* Lecture Notes in Computer Science, vol. 2072. Springer-Verlag.

Läufer, K., Baumgartner, G. & Russo, V. F. (2000) Safe structural conformance for Java. *The Comput. J.* **43**(6), 469–481.

Lee, L.-Q., Siek, J. G. & Lumsdaine, A. (1999) The Generic Graph Component Library. *Pages 399–414 of: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* ACM Press.

Leroy, X. (2000) *The Objective Caml system: Documentation and user's manual.* With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D. & Vouillon, J. (2003) *The Objective Caml documentation and user's manual.*

Litvinov, V. (1998) Contraint-based polymorphism in Cecil: towards a practical and static type system. *Pages 388–411 of: OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* ACM Press.

Magnusson, B. (1991) Code reuse considered harmful. *J. Object-oriented Program.* **4**(3).

McNamara, B. & Smaragdakis, Y. (2000) Static interfaces in C++. *First Workshop on C++ Template Programming.*

Meyer, B. (1992) *Eiffel: The language.* First edn. Prentice Hall.

Meyer, B. (1995) Static typing. *Pages 20–29 of: OOPSLA '95: Addendum to the Proceedings of the 10th Annual Conference on Object-oriented Programming Systems, Languages, and Applications.* ACM Press.

Meyer, B. (2002) The start of an Eiffel standard. *J. Object Technology*, **1**(2), 95–99. http://www.jot.fm/.

Microsoft Corporation (2002) *Generics in C#*. Part of the Gyro distribution of generics for .NET available at http://research.microsoft.com/projects/clrgen/.

Microsoft Corporation (2005) *C# version 2.0 specification, march 2005 draft*. `http://msdn.microsoft.com/vcsharp/programming/language`.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The definition of Standard ML (revised)*. MIT Press.

Myers, N. C. (1995) Traits: a new and useful template technique. *C++ report*.

Peyton Jones, S., Jones, M. & Meijer, E. (1997) Type classes: an exploration of the design space. *Proceedings of the Second Haskell Workshop*.

Peyton Jones, S., Hughes, J. *et al.* (1999) *Haskell 98: A non-strict, purely functional language*. http://www.haskell.org/onlinereport/.

Prim, R. C. (1957) Shortest connection networks and some generalizations. *Bell system technical journal*, **36**, 1389–1401.

Ramsey, N., Fisher, K. & Govereau, P. (2005) An expressive language of signatures. *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. To appear.

Rémy, D. & Vouillon, J. (1997) Objective ML: a simple object-oriented extension of ML. *Pages 40–53 of: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press.

Rémy, D. & Vouillon, J. (1998) *The reality of virtual types for free!* Unpublished note available electronically.

Siek, J. (2005) *A language for generic programming*. Ph.D. thesis, Indiana University.

Siek, J. & Lumsdaine, A. (2000) Concept checking: Binding parametric polymorphism in C++. *First Workshop on C++ Template Programming*.

Siek, J. & Lumsdaine, A. (2005a) Essential language support for generic programming. *Pages 73–84 of: PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM Press.

Siek, J. & Lumsdaine, A. (2005b) Language requirements for large-scale generic libraries. *GPCE '05: Proceedings of the Fourth International Conference on Generative Programming and Component Engineering*. To appear.

Siek, J., Lee, L.-Q. & Lumsdaine, A. (2002) *The Boost Graph Library: User guide and reference manual*. Addison-Wesley Longman.

Siek, J., Gregor, D., Garcia, R., Willcock, J., Järvi, J. & Lumsdaine, A. (2005) *Concepts for C++0x*. Tech. rept. N1758=05-0018. ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++.

Stroustrup, B. (1994) *Design and evolution of C++*. Addison-Wesley Longman.

Stroustrup, B. & Dos Reis, G. (2005) *A concept design (rev. 1)*. Tech. rept. N1782=05-0042. ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++.

Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G. & Gafter, N. (2004) Adding wildcards to the Java programming language. *Pages 1289–1296 of: SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*. ACM Press.