An Analysis of Constrained Polymorphism for Generic Programming

Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock

Open Systems Lab
Indiana University
Bloomington, IN USA
{jajarvi,lums,jsiek,jewillco}@osl.iu.edu

Abstract. Support for object-oriented programming has become an integral part of mainstream languages, and more recently generic programming has gained widespread acceptance. A natural question is how these two paradigms, and their underlying language mechanisms, should interact. One particular design option, that of using subtyping to constrain the type parameters of generic functions, has been chosen for the generics extensions to Java and C#. The leading alternative to subtype-based constraints is to use type classes, as they are called in Haskell, or concepts, as they are called in the C++ generic programming community. In this paper we argue that while object-oriented interfaces and concepts are similar in many ways, they also have subtle but important differences that make concepts more suitable for constraining polymorphism in generic programming.

1 Introduction

Generic programming is an emerging programming paradigm for writing highly reusable libraries of algorithms. The generic programming approach has been used extensively within the C++ community, in libraries such as the Standard Template Library [30, 31], Boost Graph Library [28] and Matrix Template Library [29]. Generic algorithms are parameterized with respect to the types of their arguments so that a single implementation may work on a broad class of different argument types.

For modularity, it is important for generic functions to be type checked separately from their call sites. The body of a generic function should be type checked with respect to its interface, and a call to that function should be type checked with respect to the same interface. Separate type checking helps the author of the generic function to catch errors in its interface and implementation, and more importantly, provides better error messages for incorrect uses of generic functions.

To provide separate type checking, a programming language must have a mechanism for constraining polymorphism. Several mainstream object-oriented languages with support, or proposed support, for generics, such as Java, C#, and Eiffel, implement variations of *F-bounded polymorphism* [6]. Haskell, a modern functional language, uses *type classes* [34] as the constraint mechanism for polymorphic functions. ML has parameterized modules, called functors, whose parameters are constrained by *signatures*. Other approaches include *where clauses* in CLU [23]. C++ is an example of a language

without built-in support for constraints, and which has no direct support for separate type checking: the body of a generic function is type checked at each call site.

In our recent study [14], we evaluated six mainstream programming languages with respect to their support for generic programming. Mainstream object-oriented languages did not rank highly in this evaluation; practical problems encountered include verbose code, redundant code, and difficulties in composing separately defined generic components. These problems relate to the constraint mechanisms used in the various languages. Consequently, this paper focuses on the suitability of different constraint mechanisms for use in generic programming. We analyze current manifestations of subtype-bounded polymorphism in mainstream object-oriented languages, as well as other constraint mechanisms proposed in the literature, and identify the causes of the above problems. We argue that the current implementations of subtype-based constraint mechanisms in mainstream object-oriented languages are a major hindrance to effective generic programming; it proves difficult to organize constraints into well-encapsulated abstractions. We describe how object-oriented languages could be adapted to avoid the problems mentioned above. The inspiration for the proposed changes comes from constraint mechanisms such as those in Haskell and ML, which are not affected by these problems.

2 Background

We start with a short description of generic programming, and then describe two families of type systems/language mechanisms for supporting generic programming. The first family is based on just parametric polymorphism whereas the second family is based on subtype-bounded parametric polymorphism.

2.1 Generic programming

Generic programming is a systematic approach to software reuse. In particular, it focuses on finding the most general (or abstract) formulations of algorithms and then efficiently implementing them. These two aspects, generality and efficiency, are opposing forces, which is perhaps the most challenging aspect of this practice. The goal is for a single algorithm implementation to be usable in as many situations as reasonably possible without sacrificing performance. To cover all situations with the best performance, it is often necessary to provide a small family of generic algorithms with automatic dispatching to the appropriate implementation based on the properties of the input types.

There are several ways in which an algorithm can be made more general. The simplest and most common method is to parameterize the types of the elements the algorithm operates on. For example, instead of writing a matrix multiply function that works only for matrices of ***double***, one can parameterize the function for matrices of any numeric type. Another way in which algorithms can be parameterized is on the representations of the data structures they manipulate. For example, a linear search function can be generalized to work on linked lists, arrays, or indeed any sequential data structure provided the appropriate common interface can be formulated. Yet another approach to generalization is to parameterize certain actions taken by the algorithm. For example,

in the context of graph algorithms, a breadth-first search (BFS) algorithm can invoke a user-defined callback function when tree edges are discovered. A client could use this to record the parent of each node in a BFS tree for the graph. The end result of this abstraction process should be an algorithm that places the minimum number of requirements on its input while still performing the task efficiently.

Terminology Fundamental to realizing generic algorithms is the notion of abstraction: generic algorithms are specified in terms of abstract properties of types, not in terms of particular types. In the terminology of generic programming, a *concept* is the formalization of an abstraction as a set of requirements on a type (or on several types) [1, 18]. These requirements may be semantic as well as syntactic. A concept may incorporate the requirements of another concept, in which case the first concept is said to *refine* the second. Types that meet the requirements of a concept are said to *model* the concept. Note that it is not necessarily the case that the requirements of a concept involve just one type; sometimes a concept involves multiple types and specifies their relationships.

A concept consists of four different types of requirements: associated types, function signatures, semantic constraints, and complexity guarantees. The *associated types* of a concept specify mappings from the modeling type to other collaborating types (such as the mapping from a container to the type of its elements). The function signatures specify the operations that must be implemented for the modeling type. A *syntactic concept* consists of just associated types and function signatures, whereas a *semantic concept* also includes semantic constraints and complexity guarantees [18]. At this point in the state of the art, type systems typically do not include semantic constraints and complexity guarantees. For this paper we are only concerned with syntactic concepts, so "concept" will mean "syntactic concept."

Generic programming requires some kind of polymorphism in the implementation language to allow a single algorithm to operate on many types. The remainder of this section reviews different language mechanisms and type systems that support polymorphism.

2.2 Parametric polymorphism

Generic programming has its roots in the higher-order programming style commonly used in functional languages [19]. The following ***find*** function is a simple example of this style: functions are made more general by adding function parameters and type parameters. In this example we parameterize on the ***T*** and ***Iter*** types and pass in functions for comparing elements (***eq***) and for manipulating the iterator (***next***, ***at_end***, and ***current***). This style obtains genericity using only unconstrained parametric polymorphism. For purposes of discussion we take the liberty of extending C# with polymorphic functions, function types, and type aliases as class members.

```
Iter find<Iter>(Iter iter, Iter.value_type x, ((Iter.value_type, Iter.value_type) → bool) eq,
                (Iter → Iter) next, (Iter → bool) at_end, (Iter → Iter.value_type) current)
{
  for (; !at_end(iter); iter = next(iter)) {
    Iter.value_type y = current(iter);
```

```
    if (eq(x, y))
      break;
  }
  return iter;
}

bool int_eq(int a, int b) { return a == b; }

class ArrayIterator<T> {
  typedef T value_type; ...
}

ArrayIterator<T> array_iter_next<T>(ArrayIterator<T> iter) { ... }

bool array_iter_at_end<T>(ArrayIterator<T> iter) { ... }

T array_iter_current<T>(ArrayIterator<T> iter) { ... }

void main() {
  int[] array = new int[]{1, 2, 3, 5};
  ArrayIterator<int> i(array);
  i = find(i, 2, int_eq, array_iter_next, array_iter_at_end, array_iter_current);
}
```

This example demonstrates one obvious disadvantage of the higher-order style: the large number of parameters for ***find*** makes it unwieldy to use. One solution to this problem is to introduce where clauses (various forms of which can be found in CLU [23], Theta [11], and Ada [33]). A where clause is a list of function signatures in the declaration of a generic function which are automatically looked up at each call site and implicitly passed into the function. This makes calling generic functions less verbose.

```
Iter find<Iter, T>(Iter iter, T x)
  where bool eq(T, T), Iter next(Iter), bool at_end(Iter), T current(Iter)
{ ... }

bool eq(int a, int b) { return a == b; }

class ArrayIterator<T> { ... }

ArrayIterator<T> next<T>(ArrayIterator<T> iter) { ... }

bool at_end<T>(ArrayIterator<T> iter) { ... }

T current<T>(ArrayIterator<T> iter) { ... }

void main() {
  int[] array = new int[]{1, 2, 3, 5};
  ArrayIterator<int> i(array);
  i = find(i, 2);
}
```

The addition of where clauses is not a fundamental change to the type system of the language; it is syntactic sugar for explicitly passing the function arguments.

2.3 Concepts

Similar sets of requirements often appear in many generic functions, so grouping related requirements together has software engineering benefits. For example, in a generic li-

brary such as the C++ Standard Library, all functions on sequences include requirements on their iterator parameters. Where clauses do not provide a way to group and reuse requirements. This is the role played by concepts. In the following example we create two concepts: one for expressing the comparison requirement, and one for grouping together the iterator operations. We are again using the base syntax of C#, but this time extended with concepts (we define the semantics of concepts later in this section).

```
concept Comparable<T> {
  bool eq(T, T);
}

concept Iterator<Iter> {
  type Iter.value_type; // Require an associated type

  Iter next(Iter);
  bool at_end(Iter);
  value_type current(Iter);
}

Iter find<Iter, T>(Iter iter, T x)
  where T models Comparable,
        Iter models Iterator,
        Iterator(Iter).value_type == T
{ ... }
```

A model of a concept is a set of types and a set of functions that meet the requirements of the concept. Some languages link implementations to concepts through an explicit *models declaration* (cf. Haskell instance declarations). At the call site for ***find***, for each concept requirement, a corresponding models declaration must be found.

```
int models Comparable {
  bool eq(int a, int b) { return a == b; }
}

class ArrayIterator<T> { ... }

forall<T> ArrayIterator<T> models Iterator {
  type value_type = T;
  ArrayIterator<T> next(ArrayIterator<T> iter) { ... }
  bool at_end(ArrayIterator<T> iter) { ... }
  value_type current(ArrayIterator<T> iter) { ... }
}

void main() {
  int[] array = new int[]{1, 2, 3, 5};
  ArrayIterator<int> i(array);
  i = find(i, 2);
}
```

The expression ***Iterator(Iter).value_type*** in the constraints for ***find*** accesses the ***value_type*** type definition from within the models declaration for ***Iter***. This mechanism provides a way to map from the primary types of the concept to the associated types.

Analogously to inheritance, concepts can be built from other concepts using refinement. A simple example of this is the following ***BidirectionalIterator*** concept.

concept BidirectionalIterator<Iter> : Iterator<Iter> {
  Iter prev(Iter);
}

One important observation about concepts is that they are not types. They can not be used as the type of a parameter, or to declare a variable. For the mathematically oriented, a concept is a set of multi-sorted algebras [18]. Roughly speaking, a multi-sorted algebra corresponds to a module: it is a collection of data types (the sorts) and functions (the operations of the algebra). Earlier we defined a concept as requirements on one or more types. The correspondence between these two definitions is the classic identification of a set with the predicate that specifies which elements are in the set (the elements in this case are modules).

In practice it is convenient to separate the data types of a module into two groups: the main types and the associated types. An example of this is an iterator (the main type) and its element type (an associated type). In a generic algorithm such as ***find***, a common need is the ability to obtain an associated type given the main type. A module then consists of a partial map from identifiers (names for associated types) to types $\mathrm{asc}(M) : Id \rightharpoonup Type$, and a partial map from function signatures (the name, parameter types, and result type) to function implementations $\Sigma(M) : S \rightharpoonup F$.

We formally define a concept C as a predicate on some *main types* $\vec{t}$ and a module M: $C(\vec{t}, M) = \mathcal{A} \wedge \mathcal{F} \wedge \mathcal{ST}$ where $\mathcal{A}$ is of the form $\vec{x} \subseteq \mathrm{dom}(\mathrm{asc}(M))$ (where $\vec{x}$ are the associated types required by C), $\mathcal{F}$ is of the form $\vec{s} \subseteq \mathrm{dom}(\Sigma(M))$ (where $\vec{s}$ are the function signatures required by C), and $\mathcal{ST}$ is of the form $\tau_1 = \tau_1' \wedge \cdots \wedge \tau_n = \tau_n'$ (where the τ_i and τ_i' for $i = 1 \ldots n$ are pairs of type expressions which are required to be equal). The following is the ***Iterator*** concept expressed using this notation:

Iterator(Iter,M) $\equiv$
  $\{$***value_type***$\}$ $\in$ ***dom(asc(M))*** $\wedge$
  $\{$ ***next : Iter*** $\rightarrow$ ***Iter, at_end : Iter*** $\rightarrow$ ***bool, current : Iter*** $\rightarrow$ ***asc(M)(value_type)*** $\}$ $\subseteq \Sigma(M)$

In the previous example, the body of the models declaration

forall<T> ArrayIterator<T> models Iterator {
  type value_type = T;
  ArrayIterator<T> next(ArrayIterator<T> iter) { ... }
  bool at_end(ArrayIterator<T> iter) { ... }
  value_type current(ArrayIterator<T> iter) { ... }
}

can be viewed as a parameterized module with the following set of function signatures:

ArrayIterModule $\equiv \Lambda$ ***T.***
({***(value_type, T)***}***,***
 {
  next : ArrayIterator<T> $\rightarrow$ ***ArrayIterator<T> = ...,***
  at_end : ArrayIterator<T> $\rightarrow$ ***bool = ...,***
  current : ArrayIterator<T> $\rightarrow$ ***value_type = ...***
 }***)***

So for any type ***T***, ***Iterator(ArrayIterator<T>, ArrayIterModule<T>)*** is true. We formally define that a sequence of types $\vec{t}$ together with a module M models a concept c when $c(\vec{t}, M)$ is true. We often say that a sequence of types models a concept, leaving out mention of the module of functions. This abbreviated form is written $c(\vec{t})$ and means that there is a models declaration in scope that associates a set of associated types and functions with the types $\vec{t}$ and concept c.

A concept c refines another concept c', denoted by $c \preceq c'$, if $\forall \vec{t}, m.\ c(\vec{t}, m)$ implies $c'(\vec{t}, m)$.

To describe concept-bounded types (and later subtype-bounded) we use the general setting of *qualified types* [16] to allow for a more uniform presentation. A qualified type is of the form $P \Rightarrow \tau$ where P is some predicate expression and τ is a type expression. The intuition is that if P is satisfied then $P \Rightarrow \tau$ has type τ. A qualified polymorphic type is then written

$$\forall t.\ P \Rightarrow \tau \tag{1}$$

or with multiple type parameters

$$\forall \vec{t}.\ P \Rightarrow \tau \tag{2}$$

A concept-bounded type is a qualified type where the predicates are models assertions. So concept-bounded polymorphic types have the following form.

$$\forall \vec{t}.\ c_1(\vec{t_1}) \wedge \cdots \wedge c_n(\vec{t_n}) \Rightarrow \tau \tag{3}$$

where $\vec{t_i} \subseteq \vec{t}$, the c_i's are concepts, and τ is a type expression possibly referring to types in $\vec{t}$.

The above definitions describe the structural aspects of modeling and refinement. However, languages such as Haskell and the extended C# of this paper use nominal conformance. That is, in addition to the structural properties being satisfied, there must also be explicit declarations in the program to establish the modeling and refinement relations.

Related constraint mechanisms Haskell and ML provide constraint mechanisms that share much in common with concepts. The following example, written in Haskell, groups the constraints from the previous example into type classes named ***Comparable*** and ***Iterator*** and then uses them to constrain the ***find*** (Haskell is a functional language, not object-oriented, and does not have object-oriented-style classes). In the declaration for ***find***, the ***Comparable T*** $\Rightarrow$ part is called the "context" and serves the same purpose as the CLU where clause. The ***Int*** type is made an *instance* of ***Comparable*** by providing a definition of the required operations. In generic programming terminology, we would say that ***Int*** models the ***Comparable*** concept. Note that Haskell supports multi-parameter type classes, as seen in the ***Iterator*** type class below. The syntax ***i*** $\rightarrow$ ***t*** below means that the type ***t*** is functionally dependent on ***i***, which is how we express associated types in Haskell.

```
class Comparable t where
  eq :: t → t → Bool

class Iterator i t | i → t where
  next :: i → i
  at_end :: i → Bool
  current :: i → t

find :: (Comparable t, Iterator i t) ⇒ i → t → i
find iter x =
  if (at_end iter) || eq x (current iter) then
    iter
  else
    find (next iter) x

instance Comparable Int where
  eq i j = (i == j)
```

The ***instance*** declarations can be more complex. For example, the following *conditional* instance declaration makes all lists ***Comparable***, as long as their element types are ***Comparable***:

```
instance Comparable t ⇒ Comparable [t] where
  ...
```

ML *signatures* are a structural constraint mechanism. A signature describes the public interface of a module, or *structure* as it is called in ML. A signature declares which type names, values (functions), and nested structures must appear in a structure. A signature also defines a type for each value, and a signature for each nested structure. For example, the following signature describes the requirements of ***Comparable***:

```
signature Comparable =
sig
  type ElementT
  val eq : ElementT →ElementT →bool
end
```

Any structure that provides the type ***ElementT*** and an ***eq*** function with the appropriate types conforms to this signature without any explicit instance declarations. For example:

```
structure IntCompare =
struct
  type ElementT = int
  fun eq i1 i2 = ...
end
```

2.4 Subtype-bounded polymorphism

For object-oriented languages, the subtype relation is a natural choice for constraining generic functions. This section describes the various forms of subtype-bounded polymorphism that appear in mainstream languages and in the literature.

Bounded quantification Cardelli and Wegner [7] were the first to suggest using subtyping to express constraints, and incorporated *bounded quantification* into their language named Fun. The basic idea is to use subtyping assertions in the predicate of a qualified type. For bounded quantification the predicates are restricted to the form $t \leq \sigma$ where t is a type variable and σ does not refer to t. So we have polymorphic types of the form

$$\forall t.\ t \leq \sigma \Rightarrow \tau[t] \tag{4}$$

where t is a type variable, σ is a type expression that does not refer to t, and $\tau[t]$ is a type expression τ that may refer to t.

Fun is an unusual object-oriented language in that subtyping is structural, and there are no classes or objects; it has records, variants, and recursive types. The idea of bounded quantification carries over to mainstream object-oriented languages, the main change being the kinds of types and subtyping relations in the language. Subtyping in languages such as C++, Java, and C# is between classes (or between classes and interfaces). The following is an attempt to write the ***find*** example using bounded quantification. There are two options for how to write the ***eq*** method in the ***Int*** class below. The first option results in a type error because method parameters may not be covariant (Eiffel supports covariance, but its type system is unsound [4,9]). The second option requires a downcast, opening the possibility for a run-time exception. This is an instance of the classic binary method problem [5].

```
interface Comparable {
  bool eq(Comparable);
}

Iterator find<T : Comparable>(Iterator iter, T x) { ... }

class Int : Comparable {
  bool eq(Int i) { ... } // Not a valid override
  bool eq(Comparable c) { ... } // Requires a downcast
}
```

F-bounded polymorphism Bounded quantification was generalized to *F-bounded polymorphism* by Canning et al. [6], which allows the left-hand side of a subtyping constraint to also appear in the right-hand side, thus enabling recursive constraints.

$$\forall t.\ t \leq \sigma[t] \Rightarrow \tau[t] \tag{5}$$

Types that are polymorphic in more than one type can be expressed by nesting.

$$(\forall t_1.\ t_1 \leq \sigma[t_1] \Rightarrow (\forall t_2.\ t_2 \leq \sigma[t_1, t_2] \Rightarrow (\forall t_3.\ t_3 \leq \sigma[t_1, t_2, t_3] \Rightarrow \tau[t_1, t_2, t_3])))$$

However, a constraint on type t_i may only refer to t_i and earlier type parameters.

The following example shows the ***find*** example, this time written using F-bounded polymorphism. We can now express the program without downcasts.

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

```
interface Iterator<Iter,T> {
  Iter next();
  bool at_end();
  T current();
}
Iter find<T, Iter>(Iter iter, T x)
  where T : Comparable<T>,
        Iter : Iterator<Iter,T>
{ ... }
class Int : Comparable<Int> {
  bool eq(Int i) { ... }
}
```

F-bounded polymorphism in turn was generalized to systems of mutually recursive subtyping constraints by Curtis [10, 12]. A *recursively subtype-constrained type* is of the form $P \Rightarrow \tau$ where P is a predicate of the form $\tau_1 \leq \tau_1' \wedge \cdots \wedge \tau_n \leq \tau_n'$. Then a recursively constrained polymorphic type is of the form

$$\forall \vec{t}.\ \tau_1 \leq \tau_1' \wedge \cdots \wedge \tau_n \leq \tau_n' \Rightarrow \tau \tag{6}$$

where the type variables in $\vec{t}$ can appear anywhere in the type expressions τ_i, τ_i', and τ. Recursively constrained polymorphic types, with some minor restrictions, are used in the generics extensions for Java and C#.

The following is an example of mutually recursive subtype constraints. The interface describing a graph node is parameterized on the edge type, and vice versa, and the ***breadth_first_search*** function uses the two interfaces in a mutually recursive fashion.

```
interface Node<E> {
  public List<E> out_edges();
}
interface Edge<N> {
  public N source();
  public N target();
}
public void breadth_first_search<N, E>(N n)
where N: Node<E>,
      E: Edge<N> { ... }
```

2.5 Definitions of the subtype relation

Subtype-bounded polymorphism expresses constraints based on the subtyping relation, so the expressiveness of the constraints is very much dependent on what types and subtype relations can be defined in the language. As mentioned in Section 2.4, much of the literature on bounded and F-bounded polymorphism [6, 7] used languages with records, variants, and recursive types and used a structural subtyping relation. Mainstream languages like C++, Java, and C# define subtyping as subclassing, a named subtyping relation between object types.

For a type ***B*** to be a subtype of some type ***A*** in a subtype relation that is based on structural conformance, ***B*** must have at least the same capabilities as ***A***. For example, if ***A*** is a record type, then ***B*** must have all the fields of ***A*** and the types of those fields must be subtypes of the corresponding fields in ***A***. A subtype relation based on named conformance, on the other hand, requires an explicit declaration in addition to the structural conformance requirement.

Mainstream object-oriented languages, such as C++, Java, C#, and Eiffel, unify subtyping with subclassing. The subtype relation is established at the point of definition of each class by declaring its superclasses. In particular, it is not possible to add a new supertype to an existing class without modifying the definition of the class. Mechanisms permitting such *retroactive subtyping* (or *retroactive abstraction*) declarations have been proposed and can be found in several programming languages, such as Sather [26, 27] and Cecil [8].

3 Discussion

This section discusses problems arising in object-oriented languages when attempting to follow the generic programming paradigm. Our earlier study in [14] showed that generic programming suffers from a set of distinct problems, whose cumulative effect is even more significant. As some of the symptoms, we observed verbose code in the form of excessive numbers of type parameters and constraints, awkward constructions to work around language limitations, difficulties in library maintenance, and the forced exposure of certain implementation details; the examples in [14] clearly demonstrate this.

We describe several extensions to Generic C# that lead to notably improved support for generic programming. We also describe a source-to-source translation of some of the extended features to current Generic C#.

3.1 Accessing and constraining associated types

Associated type constraints are a mechanism to encapsulate constraints on several functionally dependent types into one entity. Section 2.3 gave an example of an iterator concept and its associated type ***value_type***. As another example, consider the following two concepts specifying the requirements of a graph type. The ***IncidenceGraph*** concept requires the existence of vertex and edge associated types, and places a constraint on the edge type:

```
concept GraphEdge<Edge> {
  type Vertex;
  Vertex source(Edge);
  Vertex target(Edge);
}

concept IncidenceGraph<Graph> {
  type Vertex;
  type Edge models GraphEdge;
  Vertex == GraphEdge<Edge>.Vertex;
```

```
  type OutEdgeIterator models Iterator;
  Iterator<OutEdgeIterator>.value_type == Edge;

  OutEdgeIterator out_edges(Graph g, Vertex v);
  int out_degree(Graph g, Vertex v);
}
```

All but the most trivial concepts have associated type requirements, and thus a language for generic programming must support their expression. Of mainstream languages, ML supports this via types in structures and signatures; C++ can represent associated types as member typedefs or *traits classes* [25] but cannot express constraints on them. Java and C# do not provide a way to access and place constraints on type members of generic type parameters. However, associated types can be emulated using other language mechanisms.

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

interface IncidenceGraph {
  type Vertex;
  type Edge : GraphEdge;
  Vertex == Edge.Vertex;

  type OutEdgeIterator
    : IEnumerable<Edge>;

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

(a)

```
interface GraphEdge<Vertex1> {
  Vertex1 source();
  Vertex1 target();
}

interface IncidenceGraph<
  Vertex1, Edge1, OutEdgeIterator1>
  where
    Edge1 : GraphEdge<Vertex1>,
    OutEdgeIterator1 :
      IEnumerable<Edge1> {
  OutEdgeIterator1 out_edges(Vertex1 v);
  int out_degree(Vertex1 v);
}
```

(b)

Fig. 1. Graph concepts represented as interfaces which can contain associated types (a), and their translations to traditional interfaces (b).

A common idiom used to work around the lack of support for associated types is to add a new type parameter for each associated type. This approach is frequently used in practice. The C# ***IEnumerable<T>*** interface for iterating through containers serves as an example. When a type extends ***IEnumerable<T>*** it must bind a concrete value, the value type of the container, to the type parameter ***T***. The class ***AdjacencyList***, which extends the ***IncidenceGraph*** interface, in Figure 2(b) is an example of the same situation. The following generic function, which has ***IncidenceGraph*** as a constraint, includes

an extra type parameter for each associated type. These type parameters are used as arguments to ***IncidenceGraph*** in the constraint on the actual graph type parameter.

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator> {
  return g.out_edges(v).Current.target();
}
```

The main problem with this technique is that it fails to encapsulate associated types and constraints on them into a single concept abstraction. Every reference to a concept, whether it is being refined or used as a constraint by a generic function, needs to list all of its associated types, and possibly all constraints on those types. In a concept with several associated types, this becomes burdensome. In the experiment described in [14], the number of type parameters in generic algorithms was often more than doubled due to this effect.

A direct representation for associated types could be added to Generic C# as an extension, providing *member types* similar to those in C++. In this extension, interfaces can declare members which are placeholders for types, and place subtype constraints on these types. Classes extending these interfaces must bind concrete values to these types. As an example, Figure 1(a) shows two concepts from the domain of graphs. The ***GraphEdge*** concept declares the member type ***Vertex***. The ***IncidenceGraph*** concept has two associated types: ***Vertex*** and ***Edge***. Note the three constraints: ***Edge*** must be a subtype of ***GraphEdge***; ***Vertex*** must be the same type as the associated type, also named ***Vertex***, of ***Edge***; and ***OutEdgeIterator*** must conform to ***IEnumerable***<***Edge***>. The last constraint uses the standard ***IEnumerable*** interface which does not use the member type extension; the two styles can coexist.

This representation for associated types can straightforwardly be translated into the emulation using extra type parameters which was described earlier. Figure 1(b) shows the translated versions of the graph interfaces. In this translation, each interface containing associated types has an extra parameter added for each associated type. The subtype constraints on the associated types are converted to subtype constraints on the corresponding type parameters. In classes inheriting from such interfaces, the associated type definitions are converted to type arguments of the interfaces, as shown in Figure 2(b). Generic functions using interfaces with associated types also have an extra type parameter added for each associated type (Figure 3(b)). Within the body and constraints of a generic function, references to associated types are converted to references to the corresponding type parameters. Equality constraints between two types are handled by unifying, in the logic programming sense, the translations of the types required to be equal. For example, the type ***Vertex1*** is used both as the ***Vertex*** associated type for ***GraphEdge*** and for ***IncidenceGraph*** in Figure 1(b). Figure 2 shows the code defining two concrete classes which extend the interfaces for ***GraphEdge*** and ***IncidenceGraph***, both before and after translation. We used this translation of associated types, manually, while implementing the graph library described in [14].

The advantages of the associated type extension become evident when using interfaces to constrain type parameters of a generic algorithm. Consider the ***first_neighbor*** function in Figure 3. The function has two parameters: a graph and a vertex. Using the extension, shown in Figure 3(a), a single type parameter can describe the types and

```
class AdjListEdge : GraphEdge {
  type Vertex = int;
  ...
}
class AdjacencyList : IncidenceGraph {
  type Vertex = int;
  type Edge = AdjListEdge;
  type OutEdgeIterator =
    IEnumerable<AdjListEdge>;
  OutEdgeIterator out_edges(Vertex v) {...}
  int out_degree(Vertex v) {...}
}
```

(a)

```
class AdjListEdge : GraphEdge<int> {
  ...
}
class AdjacencyList
  : IncidenceGraph<int, AdjListEdge,
      IEnumerable<AdjListEdge> > {
  IEnumerable<AdjListEdge>
    out_edges(Vertex v) {...}
  int out_degree(Vertex v) {...}
}
```

(b)

Fig. 2. A concrete graph type which models the ***IncidenceGraph*** concept.

constraints of both of these parameters. In the translated code (Figure 3(b)), a separate type parameter is needed for each of the three associated types of the graph type.

Note that the translated code is not valid Generic C#; we are assuming that constraints on type parameters are propagated automatically from the interfaces which are used, which is not the case in the current version of Generic C#. Section 3.2 discusses this issue in more detail.

```
G.Vertex first_neighbor<G>(G g, G.Vertex v) where G : IncidenceGraph {
  return g.out_edges(v).Current.target();
}
```

(a)

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator> {
  return g.out_edges(v).Current.target();
}
```

(b)

Fig. 3. A generic algorithm using ***IncidenceGraph*** as a constraint, both with (a) and without (b) the extension.

Note that an interface that contains associated types is not a traditional object-oriented interface; in particular, such an interface is not a type. As the translation suggests, these interfaces cannot be used without providing, either implicitly or explicitly,

the values of their associated types. As a consequence, interfaces with associated types can be used as constraints on type parameters, but cannot be used as a type for variables or function parameters — uses that traditional interfaces allow. For example, the function prototype in Figure 3(a) cannot be written as:

```
IncidenceGraph.Vertex first_neighbor(IncidenceGraph g, IncidenceGraph.Vertex v);
```

The references to ***IncidenceGraph.Vertex*** are undefined; the abstract ***IncidenceGraph*** interface does not define a value for the ***Vertex*** associated type. This is a major difference between our translation and systems based on *virtual types* [24, 32]. In our translation, all associated types are looked up statically, and so the type of ***g*** is the interface ***IncidenceGraph***, not a concrete class which implements ***IncidenceGraph***. On the other hand, in systems with virtual types, member types are associated with the run-time type of a value, rather than its compile-time type; thus, the function definition above would be allowed. The virtual type systems described in [24, 32] do not provide means to express the constraints in the previous examples in type-safe manner. Ernst describes *family polymorphism* [13], a type-safe variation of virtual types, for the programming language BETA. This is a related mechanism to the extension proposed here for representing associated types in an object-oriented language. Whether family polymorphism can provide full support for associated types remains to be evaluated.

For the translation described here to work, it is important to be able to infer the values of associated types from the types bound to the main type parameters. This is not currently supported in Generic C# or Java generics. As an example of this, consider the following equivalent formulation of the ***first_neighbor*** function, which makes the use of the associated edge type more explicit:

```
G.Vertex first_neighbor<G>(G g, G.Vertex v) where G : IncidenceGraph {
  G.Edge first_edge = g.out_edges(v).Current;
  return first_edge.target();
}
```

In a call to ***first_neighbor***, a concrete graph type is bound to ***G***, and thus associated types, such as ***G.Edge***, can be resolved. In the translated version, however, it is less obvious that associated types can be inferred automatically:

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator>
{
  G_Edge first_edge = g.out_edges(v).Current;
  return first_edge.target();
}
```

The two type parameters ***G_Edge*** and ***G_OutEdgeIterator*** are not the types of any of the function arguments, and thus are not directly deducible. To infer their types, the particular graph type used as ***G*** must be examined to find its associated type definitions. The associated types are expressed as type arguments to ***IncidenceGraph*** in an inheritance declaration. Inferring the associated types from constraints is possible in most cases, including all cases generated by the translation given here, but is not supported in the current proposals for Generic C# or Java generics.

3.2 Constraint propagation

In many mainstream object-oriented languages, the constraints on the type parameters to generic types do not automatically propagate to uses of those types. For example, although a container concept may require that its iterator type model a specified iterator concept, any generic algorithm using that container concept will still need to repeat the iterator constraint. This is done for error checking: instances of an interface must always be given correct type parameters, even within the definition of a generic method. The burden of this is that the check is done when a generic method is defined, rather than when it is used, and so the generic method ends up needing to repeat the constraints of all of the interfaces which it uses.

For example, without constraint propagation, the ***first_neighbor*** function from Figure 3(a) would need to be written as:

```
G.Vertex first_neighbor<G>(G g, G.Vertex v)
  where G : IncidenceGraph,
        G.Edge : GraphEdge,
        G.Edge.Vertex == G.Vertex,
        G.OutEdgeIterator : IEnumerable<G.Edge> {
  return g.out_edges(v).Current;
}
```

The problem with constraint propagation also applies to the translated version of ***first_neighbor*** (cf. Figure 3(b)):

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator>,
        G_Edge : GraphEdge<G_Vertex>,
        G_OutEdgeIterator : IEnumerable<G_Edge> {
  return g.out_edges(v).Current;
}
```

The additional constraints in these examples merely repeat properties of the associated types of ***G*** which are already specified by the ***IncidenceGraph*** concept. This greatly increases the verbosity of generic code and adds extra dependencies on the exact contents of the ***IncidenceGraph*** interface, thus breaking the encapsulation of the concept abstraction.

This is not an inherent problem in subtype-based constraint mechanisms. For example, the Cecil language automatically propagates constraints to uses of generic types [8, § 4.2]. Constraint propagation is simple to implement: a naïve approach is to automatically copy the type parameter constraints from each interface to each of the uses of the interface.

3.3 Subclassing vs. subtyping

The subclass relation in object-oriented languages is commonly established in the class declaration, which prevents later additions to the set of superclasses of a given class. This is fairly rigid, and as many object-oriented languages unify subclassing and subtyping, the subtype relation is inflexible too. Several authors have described how this

inflexibility leads to problems in combining separately defined libraries or components, and proposed solutions. Hölzle describes problems with component integration and suggests that adding new supertypes and new methods to classes retroactively, as well as method renaming, be allowed [15]. The Half & Half system [2] allows subtyping declarations that are external to class definitions, as do the Cecil [8] and Sather [26, 27] programming languages. Aspect oriented programming systems [21], such as AspectJ [20], can provide similar functionality by allowing modification of types outside of their original definitions. Structural subtyping does not suffer from the same problems. Baumgartner and Russo [3], as well as Läufer et al. [22], suggest adding a structural subtyping mechanism to augment the nominal subtyping tied to the inheritance relation.

Constraint mechanisms more directly supporting concepts, such as Haskell type classes and ML signatures, do not exhibit the retroactive modeling problem: instance declarations in Haskell are external to types, and ML signature conformance is purely structural.

The work cited above is in the context of object-oriented programming, but the use of the subtyping relation to constrain the type parameters of generic algorithms shares the same problems. If an existing type structurally conforms to the requirements of a generic algorithm, but is not a nominal subtype of the required interface, it can not be used as the type parameter of the algorithm. Current mainstream object-oriented programming languages do not provide a mechanism for establishing this relation; types cannot retroactively be declared to be models of a given concept. This problem of retroactive modeling is described further in [14]. The research cited above has demonstrated that retroactive subtyping can be implemented for an object-oriented language.

3.4 Constraining multiple types

Some abstractions define interactions between multiple independent types, in contrast to an abstraction with a main type and several associated types. An example of this is the mathematical concept ***VectorSpace*** (more examples can be found in [17]).

```
concept VectorSpace<V, S> {
  V models Field;
  S models AdditiveGroup;
  V mult(V, S);
  V mult(S, V);
}
```

For this example, it is tempting to think that the scalar type should be an associated type of the vector type. For example, the class ***matrix***<***float***> would only have ***float*** for its scalar type. However it also makes sense to form a vector space with ***matrix***<***float***> and ***vector***<***float***> as the vector and scalar types. So in general the scalar type of a vector space is not *determined* by the vector type.

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

```
interface VectorSpace_Vector<V, S> : AdditiveGroup<V> {
  V mult(S);
}
```

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

Algorithms that require the ***VectorSpace*** concept must specify two constraints now instead of one. For example:

```
Vector linear_combination_2<Vector, Scalar>(Scalar alpha1, Vector v1,
                                            Scalar alpha2, Vector v2)
  where Vector: VectorSpace_Vector<Vector, Scalar>,
        Scalar: VectorSpace_Scalar<Vector, Scalar>
{
  return alpha1.mult(v1).add(alpha2.mult(v2));
}
```

In general, if a concept hierarchy has height n, and places constraints on two types per concept, then the number of subtype constraints needed in an algorithm is 2^n, an exponential increase in the size of the requirement specification. Concept hierarchies of height from two to five are common in practice, and we have encountered even deeper hierarchies, but 2^5 is already a large number.

The constraint propagation extension discussed in Section 3.2 ameliorates this problem. The ***VectorSpace_Scalar*** interface is attached to the ***VectorSpace_Vector*** interface by the constraint on the type parameter ***S***:

```
interface VectorSpace_Vector<V, S> : AdditiveGroup<V>
  where S : VectorSpace_Scalar<V, S>
{
  V mult(S);
}
```

This prevents the exponential increase in the number of requirements, but the interface designer must still split up concepts in an arbitrary fashion. This problem could be overcome by an automatic translation of multi-parameter concepts into several interfaces, as done above. The ***linear_combination_2*** algorithm shown above needs only a single constraint now.

```
Vector linear_combination_2<Vector, Scalar>(Scalar alpha1, Vector v1,
                                            Scalar alpha2, Vector v2)
  where Vector: VectorSpace_Vector<Vector, Scalar> {
  return alpha1.mult(v1).add(alpha2.mult(v2));
}
```

4 Conclusion

The main contribution of this paper is to provide a detailed analysis of subtype-based constraints in relation to generic programming. We survey a range of alternatives for constrained parametric polymorphism, including subtype-based constraints in object-oriented languages. We identify problems that hinder effective generic programming in mainstream object-oriented languages, and pinpoint the causes of the problems. Some

of the surveyed alternatives, such as concepts, ML signatures, and Haskell type classes, do not exhibit these problems. Based on these alternatives, we describe solutions that fit within the context of a standard object-oriented language. We describe an extension to C# that adds support for accessing and constraining associated types, constraint propagation, and multi-parameter concepts. We outline a translation of the extended features to the current Generic C# language.

Acknowledgments

We are grateful to Ronald Garcia for his comments on this paper. This work was supported by NSF grants EIA-0131354 and ACI-0219884, and by a grant from the Lilly Endowment. The fourth author was supported by a Department of Energy High Performance Computer Science Fellowship.

References

1. M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.
2. G. Baumgartner, M. Jansche, and K. Läufer. Half & Half: Multiple Dispatch and Retroactive Abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Ohio State University, 2002.
3. G. Baumgartner and V. F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice and Experience*, 25(8):863–889, August 1995.
4. K. B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Technical report, Williams College, 1996.
5. K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.
6. P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, 1989.
7. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
8. G. Chambers and the Cecil Group. *The Cecil Language: Specification and Rationale, version 3.1*. University of Washington, Computer Science and Engineering, Dec. 2002. www.cs.washington.edu/research/projects/cecil/.
9. W. R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):304–311, 1989.
10. P. Curtis. *Constrained quantification in polymorphic type analysis*. PhD thesis, Cornell University, Feb. 1990. www.parc.xerox.com/company/history/publications/bw-ps-gz/csl90-1.ps.gz.
11. M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA*, pages 156–158, 1995.
12. J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1. Elsevier, 1995.

13. E. Ernst. Family polymorphism. In *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, June 2001.
14. R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA*, Oct. 2003. To appear.
15. U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 36–55. Springer, July 1993.
16. M. P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
17. S. P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.
18. D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, July 1992.
19. A. Kershenbaum, D. Musser, and A. Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, 1988.
20. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
21. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, June 1997.
22. K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for Java. *Computer Journal*, 43(6):469–481, 2001.
23. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.
24. O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA*, pages 397–406. ACM Press, 1989.
25. N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
26. S. M. Omohundro. The Sather programming language. *Dr. Dobb's Journal*, 18(11):42–48, October 1993.
27. Sather home pages. www.icsi.berkeley.edu/~sather/.
28. J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library User Guide and Reference Manual*. Addison Wesley Professional, 2001.
29. J. G. Siek and A. Lumsdaine. A modern framework for portable high performance numerical linear algebra. In *Modern Software Tools for Scientific Computing*. Birkhäuser, 1999.
30. A. Stepanov. The Standard Template Library — how do you build an algorithm that is both generic and efficient? *Byte Magazine*, 20(10), Oct. 1995.
31. A. A. Stepanov and M. Lee. The standard template library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, Apr. 1994. (http://www.hpl.hp.com/techreports).
32. K. K. Thorup. Genericity in Java with virtual types. In *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, 1997.
33. United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983.
34. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.