Concept-Controlled Polymorphism

Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine

Open Systems Laboratory, Indiana University, Bloomington, IN 47405 USA
{jajarvi,jewillco,lums}@osl.iu.edu

Abstract. Concepts – sets of abstractions related by common requirements – have a central role in generic programming. This paper proposes a general framework for using concepts to control polymorphism in different ways. First, concepts can be used to constrain parametric polymorphism, as exemplified by type classes in Haskell. Second, concepts can be used to provide fine-grained control of function and operator overloading. Finally, generic functions can be overloaded (specialized) based on concepts, rather than simply on types. We describe a C++ implementation of a new mechanism, which we call ***enable_if***, and its role in concept-controlled polymorphism.

1 Introduction

A definition of generic programming from [17] states that its goal is the expression of algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Fundamental to realizing generic algorithms is the notion of abstraction: generic algorithms are specified in terms of abstract properties of types, not in terms of particular types. Following Stepanov and Austern, we use the term *concept* to mean the formalization of an abstraction as a set of requirements on a type (or on a set of types) [4]. Types that meet the requirements of a concept are said to *model* the concept. Ideally, a generic algorithm is defined only (and exactly) for those argument types that are models of the concepts in its interface; types that do not model the specified concept are *constrained* from use with the function.

Constraining (bounding) polymorphism is one important application of concepts to generic programming. However, concepts can be used to direct the operation of polymorphism in different ways as well, a property we refer to as *concept-controlled* polymorphism. Besides constraining parametric polymorphism, concepts can be used to provide fine-grained control of function and operator overloading. In addition, generic algorithms can be overloaded (specialized) based on concepts, rather than simply on types.

This paper applies results of research on type classes [21, 24, 41] to get closer to the ideal of constrained polymorphism. Type classes were first introduced to Haskell to control ad-hoc polymorphism, and are now found in several languages, including Gofer [19], Isabelle [43], and Mercury [18]. This paper demonstrates that type classes can to a large extent be emulated in C++ as well. Type classes are closely related to concepts in generic programming [4, 28]. Like concepts, type classes encapsulate requirements for a type, or for a set of types collectively. In C++, concepts are merely

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 228–244, 2003.
© Springer-Verlag Berlin Heidelberg 2003

part of documentation whereas type classes in Haskell are explicit representations of concepts.

The presented type class emulation builds on the work on *static interfaces* in C++ [27]. The main focus of static interfaces is in checking concept requirements for improved error diagnostics, but they also provide a mechanism for concept-based dispatching. This technique uses a generic forwarding function, which dispatches to a set of different implementation functions. Recent findings in the C++ template system let us go further in this direction, allowing concept-based overloading without any forwarding functions. In short, a class template specialization can be augmented with a static metaprogram that examines properties of the template arguments and conditionally enables or disables the specialization. Analogously, such guards can be attached to function templates.

We describe a new mechanism, which we call ***enable_if***, for supporting concept-controlled polymorphism in Section 2. Section 3 describes Haskell type classes and means for emulating them in C++. As an application of emulated type classes in C++, Section 4 suggests a replacement for the "Barton-Nackman" technique [5]. Section 5 discusses concept-based overloading using the infamous ***swap*** function as a motivating example[1]. Related approaches are described in Section 6. Finally, we discuss our conclusions in Section 7.

2 Background

Since the discovery [36] and early applications [38, 39] of C++'s template-based sublanguage for *static metaprogramming*, a notable amount of effort has been put into the development of metaprogramming tools and techniques [3, 8, 14, 37], and several C++ template libraries [15, 32, 33, 40] use static metaprogramming extensively. Consequently, C++ has become a popular and readily available platform for generic and generative programming. The primary use of static metaprogramming in generative programming is in the assembly of components; static metaprograms allow fine-grained control over how code fragments are welded together to form concrete components. In this paper, we use static metaprogramming to attain this level of fine-grained control over function overloading.

In C++, a complex set of rules determines which function definition is invoked for a particular function call expression. As a part of these rules, *argument dependent lookup* brings functions from many namespaces into the overload resolution set. Functions from namespaces that define any of the argument types, or any classes that the argument types derive from, are included. For argument types that are instances of a template, the namespaces containing the definitions of the template argument types are considered as well. As these rules apply transitively, the set of functions considered for overload resolution can contain functions defined for unrelated purposes, coincidentally sharing the same name. This can lead to compilation errors due to ambiguous function definitions residing in different, and seemingly unrelated, modules. Worse, the best matching function can be different from what the programmer intended, producing no compiler

[1] Allowing a safe mechanism for users to provide a specialized ***swap*** function to be used by standard algorithms has been a topic of long discussions in the C++ standards committee.

diagnostic but an erroneous program [1]. The term *ad-hoc polymorphism* [35] for function overloading is well justified for C++.

Sensible operation of template function overloading in C++ relies on the *SFINAE* (substitution-failure-is-not-an-error) principle [37]: if an invalid argument type or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set instead of causing a compilation error. This section describes the SFINAE principle and tools to take advantage of it, summarizing the discussion in [16].

When multiple definitions of a function template exist in a program, it is often the case that some of them are invalid for particular argument types. For example, consider the following simple function:

```
int negate(int i) { return -i; }
int i; ...; negate(i);
```

Assume further, that another overload defining **negate** for nullary functions exists in some visible namespace:

```
template <class F>
typename F::result_type negate(const F& f) { return -f(); }
```

Overload resolution must consider both definitions to select the better match. Attempting to instantiate the latter definition for a call with an argument of type **int** would result in the prototype:

```
int::result_type negate(const int&);
```

The return type **int::result_type** is invalid; **int** has no nested types. By applying the SFINAE principle, however, the function definition is removed from the overload resolution set, the error is avoided, and the former definition of **negate** is selected. A template system without SFINAE would be fragile; adding a function template that was never called could break existing working code.

Conditions where the SFINAE principle applies can be created intentionally. SFINAE is a crucial ingredient for many *compile-time reflection* tools in C++. For example, it is possible to check at compile-time whether a class has a given member type, or whether a given type is a class or not [37]. The uses of SFINAE are, however, not limited to compile-time reflection. Any function template can be accompanied with a compile-time predicate that determines whether the function is to be considered as a candidate for the best matching definition [16]. Furthermore, matching partial specializations of class templates is performed with the same set of rules as function template argument deduction. This means that a partial specialization of a class template can only match if template argument deduction does not fail, and thus the SFINAE principle can be exploited to enable and disable class template specializations [34].

The following simple template provides a controlled mechanism to create the SFINAE conditions:

```
template <bool B, class T = void>
struct enable_if { typedef T type; };

template <class T>
struct enable_if<false, T> {};
```

The ***enable_if*** template selectively includes or fails to include a member type ***type*** based on the value of the parameter ***B***. If ***B*** is true, the ***type*** member will be defined to be the ***T*** parameter to ***enable_if***. If ***B*** is false, no member will be defined. Thus, a reference to ***enable_if<B, T>::type*** is either valid or invalid depending on whether ***B*** is true or false.

Wrapping the actual return type of a function in the ***enable_if*** template adds a guard to the function. For example, this code defines a function ***foo*** whose condition is always true, and is thus always enabled; and another function ***bar*** whose condition is always false, and is thus always disabled:

```
template<class T>
typename enable_if<true, T>::type foo(T t) { return t; }

template<class T>
typename enable_if<false, T>::type bar(T t) { return t; }
```

For any copy constructible argument type, a call to ***foo*** will always succeed, while a call to ***bar*** will always cause an error, because it is not possible to create a valid instance of ***bar*** for any argument type. Instead of constant truth values, the condition can be a compile-time function that depends on the properties of one or more template arguments.

The same ***enable_if*** template can be applied to enabling and disabling class template specializations [34]. One extra template parameter is needed for the guard. Each specialization inserts the guard as the value of this extra argument. The template parameter has a default value ***void*** and thus the guard is not visible for the client of the template. In the following example, we use traits classes ***is_integral*** and ***is_float*** from the Boost type_traits library [26] as the predicates. For example:

```
template<class T, class Enable = void> class A;

template<class T>
class A<T, typename enable_if<is_integral<T>::value>::type> { ... };

template<class T>
class A<T, typename enable_if<is_float<T>::value>::type> { ... };
```

Instantiating ***A*** with any integral type matches the first specialization, whereas any floating point type matches the second one. As with function templates, the condition can be any expression that depends on the template arguments of the class, and that can be evaluated at compile-time.

3 Emulating Type Classes

3.1 Haskell Type Classes

Type classes were introduced to Haskell to allow controlled function overloading [41]. A type class definition contains a set of function signatures (and possibly their default implementations). The type class represents the collection of types for which that set of functions is defined. As an example, the following type class ***Showable*** represents the collection of all types ***t*** for which the ***show*** function is defined (the *instances* of ***Showable***):

```
class Showable t where
  show :: t → String
```

This is a simplification of the ***Show*** type class in the Haskell Prelude [23]. The presentation of the various instances of ***Showable*** is also similar to the presentation in [41].

Types are not automatically instances of ***Showable***. Instead, an explicit *instance declaration* is required:

```
instance Showable Int where
  show i = ...
```

The ellipsis stands for the definition of the ***show*** function for ***Int*** values. The instance declaration thus makes ***Int*** values ***Showable*** and attaches an implementation to the ***show*** function for ***Int***s. A single instance declaration can apply to more than one type. According to the following declaration, for any type ***t*** that is an instance of ***Showable***, lists containing elements of type ***t*** are also instances of ***Showable***:

```
instance Showable t ⇒ Showable [t] where
  show l = ...
```

Haskell constrains the type arguments of polymorphic functions using type classes. These constraints ensure that the body of the polymorphic function does not refer to any function that is not defined for the argument types. For example, the ***print*** function below outputs a value of any ***Showable*** type. The *context* ***Showable t***, before the ⇒ symbol, expresses the constraint on the argument type of the ***print*** function:

```
print :: Showable t ⇒ t → IO ()
print x = putStr (show x)
```

Type classes are a rich formalism and provide more features than these simple examples show. Functions declared in type classes can have default implementations, which are applied if no definition is given in an instance declaration. A type class ***A*** can inherit from one or more other classes: all instances of ***A*** must first be instances of its superclasses, and also meet the requirements of ***A***. Higher-order type classes act as constraints on type constructors (analogous to templates) rather than on concrete types. In addition to these features defined in Haskell 98 [23], widely available implementations include several extensions to type classes. These include *multi-parameter type classes* to allow constraints on multiple related types, and *functional dependencies* that can express a requirement that one type is accessible from another type, serving the same purpose as associated types and traits classes in C++ [22, 24].

3.2 Type Classes in C++

The C++ representation of a type class consists of two separate C++ class templates. The following two templates correspond to the Haskell version of the Showable type class (sans-serif text is used when referring to a type class abstractly, as opposed to its representation as C++ code):

```
template <class T, class Enable = void>
struct showable_traits { static const bool conforms = false; };
```

```
template <class T>
struct showable {
  BOOST_STATIC_ASSERT(showable_traits<T>::conforms);
  string show(const T& x) {return showable_traits<T>::show(x);}
};
```

The parameter ***T*** in both classes is the actual type class parameter. The first class encodes whether the type ***T*** is an instance of the type class; each specialization of this traits class must contain a boolean constant to indicate this. By convention, we name this constant ***conforms***. The ***Enable*** parameter is for compile-time predicates that enable conditional instance declarations. The second class template contains the functions that the instance types must implement. Unlike in Haskell, the existence of implementations for these functions are not checked at the point of an instance declaration.

Note that the functions are not just signatures, but define bodies that forward the calls to the implementation functions which are defined within the instance declarations. This forwarding helps to ensure that generic functions do not accidentally rely on functions not listed in type classes that constrain template arguments. The compile-time assertion (see [25] for the definition of ***BOOST_STATIC_ASSERT***) prevents instantiations on types that are not instances of the type class (according to the ***conforms*** constant). This could be extended to include an invocation of a concept checking library [27, 30, 31] to ensure that the instance declaration contains all of the required functions. Such solutions are not part of our framework yet.

The primary template ***showable_traits*** defines the ***conforms*** constant with the value ***false***. Thus, the default is that no type is an instance of Showable. As in Haskell, an explicit instance declaration is used to express that a type is an instance of a certain type class. The C++ counterparts for instance declarations are class template specializations. For example, the following specialization declares that ***int*** is Showable:

```
template <>
struct showable_traits<int> {
  static const bool conforms = true;
  string show(int x) { /* ... */ }
};
```
(3.1)

The ***conforms*** constant has the value ***true***, stating the intent that ***int*** conforms to the requirements of the Showable type class. The specialization also provides the implementation of all the required functions (just ***show*** in this example) for integers.

As in Haskell type classes, a single instance declaration can apply to more than one type. C++ can readily express sets of types with one expression, but they are limited to certain kinds of sets, such as all instances of a given template, all pointer types, and all ***const***-qualified types. Creating instance declarations that cover such sets of types requires nothing more than partial template specialization. For example, the following instance declaration makes all pointer types showable:

```
template <class T>
struct showable_traits<T*> {
  static const bool conforms = true;
  string show(T* x) { /* ... */ }
};
```

Traditional partial specialization cannot, however, express instance declarations for arbitrary sets of types. Section 3.1 showed the Haskell version of an instance declaration covering all lists where the element type is an instance of Showable. To be able to support instance declarations on such sets of types, the ***enable_if*** template can be used as a predicate to guard template specializations. This is where the ***Enable*** template parameter is needed. As a concrete example, the following instance declaration adds ***list<T>*** to Showable types, but only when the element type ***T*** is an instance of Showable:

```
template <class T>
struct showable_traits<
  list<T>,
  typename enable_if<showable_traits<T>::conforms>::type>
{                                                                    (3.2)
  static const bool conforms = true;
  string show(const list<T>& x) { /* ... */ }
};
```

Without the predicate, all types of the form ***list<T>*** would match the specialization, even lists with non-showable element types. The predicate guarantees that the specialization is only considered for lists with showable element types. The instance declaration does not say anything about lists of non-showable elements. Other instance declarations are free to make lists of certain non-showable types showable using a different implementation of the ***show*** function.

The above definitions create a representation of the Showable type class, and make ***int***, all pointers, and all lists with showable element types showable. The Showable type class can now be used to constrain parameters of function templates. The C++ equivalent of the Haskell ***print*** function shown in Section 3.1 is defined as:

```
template <class T>
typename enable_if<showable_traits<T>::conforms, void>::type
print(const T& t) {
  typedef showable<T> st;
  std::cout << st::show(t);
}
```

The enabler causes the function to exist only for types that are showable. An attempt to call it with a non-showable type causes a "no function found" error rather than an error in the function's body. The function gains access to the functions from the Showable type class by instantiating the ***showable*** class with ***T*** as the argument. The ***print*** function uses a *qualified name* in the ***show*** invocation to ensure that the ***show*** function defined in the instance declaration ***showable_traits<T>*** is called. Thus we can avoid all the subtleties of C++ argument dependent lookup of functions [1].

The ***print*** function can be called without any extra syntax for the user:

```
int a = 5; print(a);
list<int> b; print(b);
```

In the first call to ***print***, ***showable*** will be instantiated with ***int*** as the showable type, and so the instance declaration in Listing (3.1) will be used to define the ***show*** function. In the second call, ***int*** is again a showable type, and so ***list<int>*** is also showable. In this

case, the definition of ***show*** for lists in Listing (3.2) will be used, and each element will be printed using the definition of ***show*** for integers.

The C++ type class simulation can readily implement many of the advanced features of type classes discussed in Section 3.1. The class templates can have more than one template parameter, thus allowing multi-parameter type classes. It is also possible to use template template parameters to allow the equivalent of higher-order type classes. Defaults can be provided for the implementations of type class functions by providing a class template containing default implementations from which the instance declarations can inherit. Type classes can also inherit from other type classes, which requires that each instance of the subclass first be an instance of the superclass. This can be emulated in C++ by adding another static assertion into the subclass's wrapper class, ensuring that instance declarations for the superclasses have been created. This check occurs only when the instance declaration is used; there does not seem to be a way to ensure that a type is an instance of the superclasses at the time of an instance declaration.

The original type class proposal [41] outlined an implementation where type classes and their instance declarations are translated into "method dictionaries" and functions to access the methods in the dictionary. The C++ type classes are a direct translation of this dictionary technique. Unlike the original Haskell version, however, the C++ type class simulation does not require any extra run-time computation.

3.3 Discussion

The type class emulation in C++ is only partial. Most importantly, C++ type classes lack structural conformance checking. At the point of an instance declaration, no checks are made to ensure that the required functions exist, or that they conform to the required signatures. Errors are thus caught later, at the call sites to the functions defined in the type class. This decreases the quality of error diagnostics. There are library solutions for structural conformance checking [27, 31] which, however, are not yet incorporated into our type classes. Conditional instance declarations, however, remain a problem. Concept checking occurs at the time of instantiating a template with concrete types; conditional instance declarations describe properties of unbounded sets of types, not any particular type, resulting in an unlimited number of concrete instances to check.

The C++ type class emulation is syntactically heavier than Haskell type classes. A generic function in Haskell can call functions defined in other type classes without any extra qualification, whereas in C++ the type class must be explicitly specified. Furthermore, Haskell can infer the type class constraints for a generic function from the uses of the parameters in the body of the function.

C++ type classes provide features not found in Haskell type classes. The conditions on function and traits class overloads in C++ can be any boolean expressions, including negations and disjunctions. Thus, a function overload can be enabled only when some type is *not* an instance of a type class, as demonstrated by the examples in Sections 4 and 5. A C++ type class can contain other kinds of members besides function signatures, including member types and member templates.

In Haskell, each function name must only appear in a single type class, which is a prerequisite for being able to infer type class constraints from expressions in the bodies of functions. In C++, the same function name can occur in more than one type class, even

within the same module of the program. Constraints cannot obviously be inferred but, on the other hand, unrelated parts of a program do not need to handle possible name conflicts between function names in type classes. Allowing overlapping type classes adds to the flexibility of the C++ type class emulation. For example, assume the type classes Less Than Comparable with the ***less*** function, Greater Than Comparable with the ***greater*** function, and Minimizable with the ***min*** function. The ***min*** function can be implemented using either the ***less*** or ***greater*** function, but instance declarations to express this cannot be defined in Haskell; a type could be an instance of both Less Than Comparable and Greater Than Comparable, leading to an ambiguity.

Type classes are an explicit, though partial, representation of concepts in generic programming. The relationship of types modeling concepts is represented by the instance relationship between types and type classes, and type class inheritance corresponds to concept refinement. Haskell type classes can only describe requirements on functions, whereas C++ type classes can contain types and class templates. This gives a direct representation for associated types.

4 A Replacement for the "Barton-Nackman Trick"

The "Barton-Nackman trick" [5], also known as the *Curiously recurring template pattern* [7], enables a strategy for creating controlled overloads for functions and operators. In this section we show how type classes provide a simpler and safer alternative to achieve the benefits of the Barton-Nackman technique.

In the Barton-Nackman technique, a template class derives from an instantiation of another template, passing itself as a template parameter to that other template as follows:

```
class A : public base<A> { ... };
```

The important application of this technique is to give a common "base template" for a set of template classes. One can overload operators and other non-member functions for the common base template; the derived classes will match the definitions. Because it avoids run-time dispatching, this technique is often used in the numerical domain [11–13, 42],

The usage of the Barton-Nackman trick is best explained with an example, which we take from the domain of linear algebra. Matrix types (***dense_matrix***, ***diagonal_matrix***, and ***triangular_matrix***) represent different kinds of matrices. Each of these matrix types has a common interface which can be exploited to define functions and operators that work for any matrix type. In this example, our interface is just one function. We overload the function call operator that gives the syntax ***A(i,j)*** for accessing the element at row ***i*** and column ***j*** of matrix ***A***. This interface is defined in the common base template ***matrix***:

```
template <class Derived>
class matrix {
public:
  double operator()(int i, int j) const {
    return static_cast<const Derived&>(*this)(i,j);
  }
};
```

The ***matrix*** template will always be instantiated in such a way that the template parameter ***Derived*** is the type of a derived class. Note the static cast in the definition of ***operator()***. As long as ***Derived*** is the actual type of the object, this is a safe downcast to a derived class. The function call operator thus invokes the function call operator of the derived class without dynamic dispatching.

Each specialized matrix type must pass itself as a template argument to the ***matrix*** template and define the actual implementation of ***operator()***:

```
class dense_matrix : public matrix<dense_matrix> {
public:
  double operator()(int i, int j) { ... }
  ...
};

class diagonal_matrix : public matrix<diagonal_matrix> {
public:
  double operator()(int i, int j) { ... }
  ...
};

class triangular_matrix : public matrix<triangular_matrix> {
public:
  double operator()(int i, int j) { ... }
  ...
};
```

With this arrangement, one can overload generic functions for the ***matrix*** template, instead of separately for each matrix type. For example, the multiplication operator could be defined as:

```
template<class A, class B>
typename product_traits<A, B>::type
operator*(const matrix<A>& a, const matrix<B>& b);
```

The ***product_traits*** template is a traits class to determine the result type of multiplying matrices of types ***A*** and ***B***. Such traits classes for deducing return types of operations are common in C++ template libraries [15, 40, 42].

We can observe many benefits in this approach. The run-time overhead of virtual function calls is avoided. Several implementation types can be grouped under a common base template, allowing one operator implementation to cover many separate types. We can also identify several problems with the approach:

- One needs to be careful not to slice objects. Taking the matrix arguments by copy in the above ***operator****∗* function would not copy the whole object, but only the base class ***matrix***. Thus some data would be lost, leading to undefined behavior at run-time when the object was cast to the actual matrix type in the function call operator of the ***matrix*** class.
- It is difficult to get the desired overloading behavior if some parameters of a function are instances of "Barton-Nackman powered" types, but others are not. For example, one might want to define multiplication between two matrices, and additionally between a matrix and a scalar with the scalar as either the left or right

argument. The straightforward solution is to define three function templates (we leave the return types unspecified):

```
template<class A, class B>
... operator*(const matrix<A>& a, const B& b);

template<class A, class B>
... operator*(const A& a, const matrix<B>& b);

template<class A, class B>
... operator*(const matrix<A>& m1, const matrix<B>& m2);
```

The following code demonstrates why this approach does not work:

```
diagonal_matrix A; dense_matrix B;
A * B; // error, ambiguous call
```

The third function is not the best match; instead, the call is ambiguous. The first two definitions are both better than the third definition, but neither is better than the other. Thus, a compiler error occurs. Even though the type ***matrix*<*A*>** is more specialized than ***A***, the actual argument type is not ***matrix*<*A*>** but ***diagonal_matrix***, which derives from ***matrix*<*A*>** where ***A*** is ***diagonal_matrix***. Therefore, matching ***diagonal_matrix*** with ***matrix*<*A*>** requires a cast, whereas matching it with ***A*** does not, making the latter a better match. Thus, the second definition provides a better match for the first argument, and the first definition provides a better match for the second argument; this makes the call ambiguous.

There are two immediate workarounds: providing explicit overloads for all common scalar types, or overloading for the element types of the matrix arguments. The first solution is tedious and not generic because one cannot know all the possible (user-defined) types that might be used as scalars. The second solution is limiting, as it prevents sensible combinations of argument types, such as multiplying a matrix with elements of type ***double*** with a scalar of type ***int***.

– The curiously recurring template pattern is quite a mind-teaser; it significantly adds to the complexity of the code.

Type classes and the ***enable_if*** template solve the above problems. We rewrite our matrix example using type classes. First, all matrix types must be instances of the following type class:

```
template <class T, class Enable = void>
struct matrix_traits {
  static const bool conforms = false;
};

template <class T>
struct matrix {
  BOOST_STATIC_ASSERT(matrix_traits<T>::conforms);
  static double index(const T& M, int i, int j) {
    return matrix_traits<T>::index(M, i, j);
  }
};
```

The element access operator ***operator()*** in the type class is renamed to ***index***, because type classes cannot contain non-static member functions. The concrete matrix classes can still use ***operator()***; instance declarations can forward the call to an arbitrary function or operator. The instance declaration for ***dense_matrix*** is (the other matrix types are analogous):

```
class dense_matrix {
  double operator()(int i, int j) { ... }
};

template <>
struct matrix_traits<dense_matrix> {
  static const bool conforms = true;
  static double index(const dense_matrix& M, int i, int j) {
    return M(i,j);
  }
};
```

Now the multiplication operator can be defined on any pair of argument types where at least one is a matrix type. Using type classes, one definition is enough to cover exactly the valid cases:

```
template<class A, class B>
typename enable_if<
  matrix_traits<A>::conforms || matrix_traits<B>::conforms,
  typename product_traits<A, B>::type>::type
operator*(const A& a, const B& b);
```

This operator is only enabled if at least one of the arguments is an instance of the Matrix type class.

Regarding the three drawbacks of overloading based on the Barton-Nackman trick, type classes solve the ambiguity problem. Slicing is also not a problem as ***A*** and ***B*** are unconstrained template parameters that are deduced to the actual types of the function arguments, rather than base classes of these types. Hence, parameter passing could safely be by copy as well. Third, the code is more direct.

Note also, that adding new types to the type class is non-intrusive. The class ***dense_matrix*** contains no reference to the representation of the Matrix type class, and it does not even use the function names required by the type class. Any matrix type supporting element access in some form can be added to the type class without modifications to the definition of the matrix type, giving it all the operators and functions defined for the type class.

5 Concept-Based Overloading

Function enabler predicates can be arbitrary boolean expressions. Functions can thus be overloaded for types that are not members of a given type class. For example, suppose that there is a generic ***swap*** function that works for any type supporting assignment. Certain types can also define a specialized ***swap*** for better efficiency. The traditional approach to this problem, used in the C++ Standard Library, is to define a generic ***swap***

function that uses assignment, and to provide the specialized implementations as overloads. Without ***enable_if*** and type class emulation, overloading is limited. For example, it is not possible to define a ***swap*** function for all types that derive from a particular class; the generic ***swap*** is a better match if the overloaded function requires a conversion from a derived class to a base class (see section 4). Type classes allow us to express the exact overloading rules.

First, two type classes are defined. Assignable represents types with assignment and copy construction (as in the Standard Library), and User Swappable is for types that overload the generic ***swap*** function:

```
template <class T, class Enable = void>
struct assignable_traits { static const bool conforms = false; };

template <class T>
struct assignable {
  BOOST_STATIC_ASSERT((assignable_traits<T>::conforms));
};

template <class T, class Enable = void>
struct user_swappable_traits { static const bool conforms = false; };

template <class T>
struct user_swappable {
  BOOST_STATIC_ASSERT(user_swappable_traits<T>::conforms);
  static void swap(T& a, T& b) {
    user_swappable_traits<T>::swap(a,b);
  }
};
```

The Assignable requirements are assignment and copy construction, which are not subject to overload resolution, and thus need not be routed via the type class mechanism.

Second, two overloaded definitions of ***generic_swap*** are provided. The first is for types that are instances of User Swappable. The function forwards calls to the ***swap*** function defined in the User Swappable type class:

```
template <class T>
typename enable_if<user_swappable_traits<T>::conforms, void>::type
generic_swap(T& a, T& b) {
  user_swappable<T>::swap(a,b);
}
```

The second overload is used for types which are instances of Assignable but not instances of User Swappable. The exclusion is needed to direct types that are both Assignable and User Swappable to the customized ***swap***:

```
template <class T>
typename enable_if<
  assignable_traits<T>::conforms && !user_swappable_traits<T>::conforms,
  void>::type
generic_swap(T& a, T& b) {
  T temp(a); a = b; b = temp;
}
```

Thus, the ***generic_swap*** function can be defined in the most efficient way possible for each type. There is no fear of overload resolution accidentally picking a function other than that the programmer intended.

6 Related Approaches

Our type class approach shares many of the goals with previous work on concept checking, which allows earlier detection of invalid template parameters, and more direct error messages. In particular, the work of McNamara and Smaragdakis uses a system of named conformance using traits classes, as well as structural concept checking [27]. Their work also includes a technique for concept-based dispatching, which is a useful alternative to the enabler-based technique used in this paper. The concept checking library by Siek and Lumsdaine [31] provides pure structural conformance checking, and allows arbitrary expressions as requirements. Either of the structural conformance checking approaches can be added to a type class implementation in the future to check that instance declarations for type classes contain all of the required functions. The work on C++ *signatures* [6] provided a mechanism to specify a set of required member prototypes as an interface as an extension to the GNU C++ compiler. The signature extension was originally intended for object-oriented programming, but could probably serve as a useful form of static interfaces for concept-based overloading as well. Early work suggesting concept-like structures as extensions to C or C++ includes Kind-C++ [29] and Cforall [10].

An alternative to the curiously recurring template pattern is to use friend declarations to force the instantiation of particular non-member functions. The Boost.Operators library applies this technique to provide default definitions for common operators (in terms of other operators) [2]. This technique allows non-member functions to be defined for particular types, but not easily for more than one type at a time, because it normally relies on explicit instantiation of templates.

Tag dispatching, a form of concept-based dispatching is a relatively common technique in C++. This involves a generic implementation of a function which looks up a particular *tag* type in a traits class and passes an object of this tag type together with its original arguments to a set of implementation functions. Concept-based overloading with the ***enable_if*** template avoids some of the problems of tag dispatching. The dispatcher function must be applicable for all tags, e.g., its return type expression must compute the return type for all cases. Furthermore, the language imposes restrictions on forwarding functions. It is not possible to define a generic forwarding function that accepts both non-const lvalues and rvalues and forwards them unchanged to another function [9]. Tag dispatching commonly uses inheritance relationships among the tags to implement a priority for each overload. McNamara and Smaragdakis proposed a different priority system, which uses a sequence of compile-time ***if*** statements to select among enumeration values to determine which implementation to use [27]. Our approach is closer to the latter. The ***enable_if*** approach allows the dispatching logic to be defined within the function definitions, instead of a separate rule set.

Lastly, *qualified types* present a theory for parametric polymorphism constrained using an arbitrary predicate [20]. This work also includes mappings from predicates to

program structures, such as the function definitions in a type class instance declaration, that show why a particular predicate is true. This approach is somewhat similar to the use of a traits class by a generic algorithm to look up type class function definitions for a particular instance. On the other hand, [20] does not have sets of functions which are overloaded based on predicate values, as can be done in C++.

7 Conclusion

Concepts are an important idea in generic programming, as they provide the main mechanism for constraining parameters to generic algorithms. Because concepts do not exist explicitly in C++, however, there is no easy way to select function overloads based on concept modeling relationships. This paper provides a technique for controlling function overload resolution based on concept conformance.

Type classes are an existing approach, used in several languages, for concept representation. A type class provides an encapsulation of a set of functions on a particular type, and can thus be seen as an explicit representation of a concept. Instance declarations add types into type classes; they declare that a type models the concept described by the type class. Instance declarations are non-intrusive; a type can be made to be an instance of a type class without modifying the type's definition.

Building on previous work on static interfaces [27] and recent template programming techniques [16], this paper describes a framework for emulating type classes in C++ to provide concept-controlled overloading. Generic functions can thus be defined for exactly those types that model the required concepts. Definitions cannot cause ambiguities with functions overloaded for different concepts. As a concrete application of the type class emulation, we described an improved alternative for library designs based on the curiously recurring template pattern.

The proposed type class emulation does not provide full-fledged support for type classes. Most notably, structural conformance is not enforced at the point of instance declarations. Leveraging on previous work on C++ concept checking, partial support for structural conformance can be added, but conditional instance declarations remain as a problem.

Acknowledgments

We are grateful to Howard Hinnant, Jeremy Siek, and Mat Marcus for their valuable comments and help on this work. This work was supported by NSF grants EIA-0131354 and ACI-0219884, and by a grant from the Lilly Endowment.

References

1. D. Abrahams. Qualified namespaces. Technical Report N1408=02-0066, ISO Programming Language C++ Project, November 2002. `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1408.html`.
2. D. Abrahams and J. Siek. The Boost Operators library. `www.boost.org`, 2002.

3. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley, 2001.
4. M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.
5. J. Barton and L. Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.
6. 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.
7. J. O. Coplien. Curiously recurring template patterns. *C++ Report*, pages 24–27, Feb. 1995.
8. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.
9. P. Dimov, H. Hinnant, and D. Abrahams. The forwarding problem: Arguments. Technical Report N1385=02-0043, ISO Programming Language C++ Project, September 2002. `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1385.html`.
10. G. Ditchfield. *Contextual Polymorphism*. PhD thesis, University of Waterloo, 1994.
11. G. Furnish. Disambiguated glommable expression templates. *Computers in Physics*, 11(3):263–269, May/June 1997.
12. G. Furnish. Container-free numerical algorithms in C++. *Computers in Physics*, 12(3):258–266, May/June 1998.
13. G. Furnish. Disambiguated glommable expression templates revisited. *C++ Report*, May 2000.
14. A. Gurtovoy and D. Abrahams. The Boost C++ metaprogramming library. `www.boost.org/libs/mpl`, 2002.
15. J. Järvi, G. Powell, and A. Lumsdaine. The Lambda Library: unnamed functions in C++. *Software – Practice and Experience*, 33:259–291, 2003.
16. J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003.
17. M. Jazayeri, R. Loos, and D. R. Musser, editors. *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*. Springer, 2000.
18. D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. In *Proceedings of the Twenty-third Australasian Computer Science Conference*, pages 128–137, January/February 2000.
19. M. P. Jones. *Gofer*. University of Oxford, 1992. Documentation distributed with Gofer source.
20. M. P. Jones. A theory of qualified types. In B. Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., Feb. 1992.
21. M. P. Jones. A system of constructor classes: overloading and implicit higher -order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture*, pages 52–61, New York, N.Y., 1993. ACM Press.
22. M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, number 1782 in LNCS, pages 230–244. Springer-Verlag, March 2000.
23. S. P. Jones, J. Hughes, et al. *Haskell 98: A Non-strict, Purely Functional Language*, February 1999. `http://www.haskell.org/onlinereport/`.
24. S. P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space, June 1997.
25. J. Maddock. The Boost static assertion library. `www.boost.org/libs/static_assert`, 2000.
26. J. Maddock, S. Cleary, et al. The Boost type_traits library. `www.boost.org/libs/type_traits`, 2002.

27. B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.
28. D. R. Musser and A. Stepanov. Generic programming. In *ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 1988.
29. P. Scheyen, S. Yu, M. Zhang, and Q. Zhuang. Introducing KINDs to C++. Technical Report 366, University of Western Ontario, London, Ontario, May 1993. `http://www.csd.uwo.ca/~pete/kc.ps`.
30. J. Siek. *Boost Concept Check Library*. Boost, 2000. `www.boost.org/libs/concept_check/`.
31. J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.
32. J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library User Guide and Reference Manual*. Addison Wesley Professional, 2001.
33. 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.
34. R. Smith. A default_constructible traits. Thread of Usenet articles in `comp.sci.lang.c++.moderated`, Jan. 2002.
35. C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes for the International Summer School in Computer Programming*. August 1967. As cited in [41].
36. E. Unruh. Prime number computation. Distributed in the ANSI X3J16-94-0075/ISO WG21-426 meeting, 1994.
37. D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.
38. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
39. T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
40. T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
41. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.
42. J. Walter and M. Koch. *The Boost uBLAS Library*. Boost, 2002. `www.boost.org/libs/numeric`.
43. M. Wenzel. Using axiomatic type classes in Isabelle (manual), 1995. `www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`.