Improving concept checking in Boost

István Zólyomi and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{scamel, gsd}@elte.hu

Abstract. To ensure the correctness of template based constructions in C++, constraints on template parameters are especially useful. Though C++ does not directly support checking requirements on template parameters, several solutions address this problem. Boost provides a comprehensive concept checking library that could be further advanced in many areas. In this paper we propose a different structure and an extended feature set to improve the Boost concept checking library into a general template introspection library. Our recommended solution takes the advantages of previous solutions, such as REQUIRE-like macros and static interfaces.
On the lowest level, we suggest an orthogonal construction of elementary concepts instead of a predefined set of complex constraints. These basic concepts should return a boolean results instead of aborting compilation immediately, thus any metaprogramming tool could use the result of the check. Based on these building blocks, it is possible to express highly complex constraints using arbitrary logical expressions instead of the implicit conjunction used in all current concept checking libraries.
In this article we introduce a possible implementation detailed to the source-level. Our implementation is non-intrusive, relies only on standard C++ language features and implies no runtime overhead.
Note that this is a shortened version of our paper 'Towards a general template introspection library' for GPCE'04 which can be downloaded from `http://gsd.web.elte.hu/Publications/GPCE_04/introspection.pdf`.

1 Introduction

Generic programming in C++ is characterized by the use of template parameters to represent abstract data types (or "concepts"). However, the C++ language itself does not provide a mechanism for the writer of a class or function template to explicitly state what concept the user-supplied template argument should model (or conform to).

The Boost Concept Checking Library provides:

- A mechanism for inserting compile-time checks of template parameters.
- A framework for specifying concept requirements though concept checking classes.
- A mechanism for verifying that concept requirements cover the template.

– A suite of concept checking classes and archetype classes that match the concept requirements in the C++ Standard Library.

Templates in C++ have many unique properties unlike other constructs of the language. By definition, every part of the template is instantiated only when used in the code. Unfortunately, this may cause a surprising behavior of our code during the development process. For instance, if we add a legal method call on an object of a template class, our previously accepted code may not compile anymore. This is a result of lazy instantiation: defects in the parameter type of the template are encountered only during instantiation of the required feature.

There is no builtin language support in C++ to ensure that features of a template type parameter exist, template arguments are not constrained in any way. Instead, all type checking is postponed to template instantiation time. The lack of language support is intentional (see [7]). According to Stroustrup, the flexibility of the C++ template construct makes concepts unnecessary. He considered concept checking to be even harmful. Indeed, lazy instantiation strategy allows a larger number of types to be used as parameter for a given template: our code will be valid (also for otherwise illegal types) as long as we do not try to instantiate any nonexisting member of that parameter type.

To improve reliability and maintainance of libraries, there were heavy efforts to implement concept checking on template parameters since the introduction of templates in C++. It is especially essential to detect wrong or incomplete implementations in the Standard Template Library, where some limited solutions are widely used, e.g. for iterators. These are based on enforcing the instantiation of templates manually, thus drawbacks of lazy instantiation can be avoided. This forms the base also for the current solution in Boost. Further in the article, we will refer to this solution as REQUIRE-like macros or traditional concept checking.

Another approach was presented by Smaragdakis and McNamara [1]. They introduced a framework called static interfaces, which is based on explicitly specifying concepts that a class conforms to. This is very similar to interfaces in e.g. Java, but has the advantage that checks do not raise errors, but return compile time constants which can be inspected and used later in the code.

In this paper we aim to provide a general and comprehensive, non-intrusive framework for expressing orthogonal basic concepts in C++. Our checks result compile time constants instead of compile errors, thus allowing metaprogramming techniques to be used. The structure of our framework is intended to be orthogonal. Later, based on these building blocks we present a way of concept checking that tries to take the advantages of both traditional concepts and static interfaces.

2 Introspection library design

A standard, well designed and comprehensive concept checking library would largely increase the chance of early detection of many conceptional errors in our program design. Despite the heavy efforts, there are still many deficiencies in current works on concept checking.

Firstly, most concept checking libraries (e.g. boost::concept [9]) raise compile time errors when the inspected type does not meet its requirements. Introspection and feature detection on type parameters (returning compile time constants instead of raising errors) would allow us to use compile time adaptation or any other metaprogramming algorithms, e.g. utilizing `boost::mpl` (see [10]). As an advantage of compile time adaptation techniques, e.g. a container would be able to store comparable types in a binary tree for efficient access, while it could store other types in a vector. Algorithmic strategies also could be decided, e.g. a sort method could use quicksort for random access containers, while a merge sort could be used otherwise.

Furthermore, based on boolean results, we would be able to express relationship between existing concepts using arbitrary logical operations. Note that current libraries use an implicit *and* connection between all conditions. For example, a type could be serialized to `cout` if it has `operator<<` *or* member function `print()`. Another example can be a type having *no* public constructor (e.g. singletons).

Secondly, most of the previous works were concentrating on checking particular concepts, but no comprehensive work was made on implementing elementary concepts themselves. Most examples verify the existence of a member type (e.g. `T::iterator`) or a simple function (e.g. comparision operator) in a type parameter. We can easily implement such checks for a single function, but without reusable basic concepts, they must be completely rewritten for any other function, even similar ones. (E.g. concepts EqualityCheckable and LessThanComparable are verifying operators with the same signature, but using current libraries, they both have to be written from scratch if they are not already implemented).

Thirdly, despite having several concept libraries, there are many concepts that seemingly cannot be implemented in C++. Such a concept is already mentioned above, when a type should have *no* public constructor. Note that Boost is able to require its existence, but negation of this condition (e.g. for singletons) is impossible. We have to address to implement as many basic concepts as possible to provide a complete, orthogonal set of basic concepts.

Therefore, we suppose the following strategies for a well-designed concept library:

1. Introspection of code and actions based on check results should be separated: check failures should not be bound to aborting compilation. Instead, an elementary action should be provided to interrupt compilation (like `BOOST_STATIC_ASSERT`) with custom error message if needed. Of course, any other action also could be triggered, e.g. raise warnings or flag portability issues, etc.
2. Concept checking should be factorized to orthogonal, elementary conditions. We should give tools for constructing real life, compound concepts from such basic conditions.
3. The library should be non-intrusive and extensible.

3 Elementary conditions

In this section we recommend a general solution for several basic conditions. Note that many specific basic checks already have an appropriate solution. These checks vary from checking the modifiers of the type of a variable (pointer, reference, const, etc.) to verifying the presence of a nested type in a class. We do not intend to reinvent the wheel. Instead, we concentrate on concepts that still do not have a comprehensive or general solution.

The following set of basic concepts is aimed to be orthogonal. Having a type parameter, we have language support to use the type itself, reference to one of its nested types or its members. Accordingly, we support the following atomic concepts:

- Constraints on the type, e.g. size, modifiers, etc[1]
- Existence of a nested name for
  - nested types
  - members
- For an existing name, the exact type for that name for
  - nested types
  - member functions (both static and non-static)
  - attributes (both static and non-static)

We can see that this list is far from being complete. Because of difficulties of implementation, we are still unable to checks if a type is abstract, if a function is virtual, etc. There are many concepts that would be useful to have, but seemingly impossible to implement using current language features.

3.1 Attributes

Solutions based on partial template specialization exist for checking whether a type is reference or const, discussed in [4] and [3]. In a similar manner we can check the *exact type* of an attribute. In this part we introduce our solution for this problem, which is based on function overloading instead of partial specialization.

Based on techniques discussed above, we can check the exact type for any member (being static or non-static member) in the following way:

```
template <class VarType>
struct Attribute
{
    // --- Check static member
    static Yes Static(VarType*);

    // --- Rescue for static member
    static No Static(...);
```

[1] These constraints already have an appropriate solution presented in several works, e.g. [4] and [3], hence we do not discuss them here.

```
    // --- Check non-static member
    template <class Class>
    static Yes NonStatic(VarType Class::*);

    // --- Rescue functions for non-static member
    static No NonStatic(...);
    template <class> static No NonStatic(...);
};

// --- Example of usage (results false)
bool result = CONFORMS( Attribute<int>::NonStatic( &list<int>::size ) );
```

The functionality of `Attribute` consists of two main parts: checking static and non-static members[2]. To understand how the above described programming techniques work altogether, let us explain the compilation steps of the usage example in the last line:

1. Classes `Attribute<int>` and `list<int>` are instantiated.
2. A member pointer is set to `list<int>::size`. However, it still has a currently unknown type since it can be either a member function or a data member. (Note that if `size` is a static member, a conventional pointer is gained instead)
3. `Attribute<int>::NonStatic()` is chosen according to overloading rules. If type of the pointed member matches the type parameter of the `Attribute` template (actually `int`), our template function is preferred; otherwise the `NonStatic(...)` rescue function is found.
4. The `sizeof` operator is applyied on the result type of the previous function call by the `CONFORMS` macro, while the function itself is not actually called. We gain the size of class `Yes` or `No`.
5. The result size is checked and a compile time boolean constant is finally achieved.

For a deeper understanding of class `Attribute`, we introduce all different functionalities of the class through examples.

```
struct Base { int var; };
struct Derived : public Base {};

// --- Possible forms of calls (without CONFORMS to save space)
Attribute<const int>::Static( &Derived::var );    // --- results No
Attribute<int>::NonStatic( &Derived::var );       // --- results Yes
Attribute<int>::NonStatic<Base>( &Derived::var ); // --- results Yes
```

Function `Static()` returns true only if the parameter is a static member of its class, non-static members are checked the same way by `NonStatic()`. However, this does not limit the usability of our class: if we do not care whether

[2] Global and namespace variables (and later, functions) could also be checked using function `Static()`.

the member is static or non-static, we can check both and connect the results with a *logical or* (e.g. `operator ||`).

Function `NonStatic()` has an interesting feature, shown in the last two examples. Since it is a template function, we do not need to specify its type parameter, it is automatically deduced by the compiler. We allow the examined attribute to be a member of *any* class this way. Though we do not have to, we may explicitly specify the type parameter of `NonStatic()`. In this case we check whether the inspected attribute is a member of the *specified* class.

Note that as a consequence of our implementation technique, this solution has an important property: we have to specify all type modifiers when inspecting types, because they are part of the exact type to be checked. If we check whether an attribute of type `const int` is type of `int`, we get false as result. We must always specify the *exact* type to be checked.

3.2 Implement attributes, get functions for free

Checking the exact type of functions is a more complicated, but still very similar problem to checking attributes as in 3.1. Functions have a more complex type: they have a return type, a signature and may have several qualifiers (`const`, etc). However, syntaxes for defining the type of a data member and a member function are similar and closely related. Here, we can make a great advantage of this fact: the exact type of a function can be inspected using the very same method as with attributes. We use only a typedef on our previous `Attribute` class to create `Function` and change nothing inside the class. Now we can make the following checks:

```
struct Base {
    static string classId();
    double calc(double);
};

struct Derived : public Base {
    void f(int, int);
};

// --- All examples return type Yes
Function<string ()>::Static( &Derived::classID );
Function<void (int,int)>::NonStatic( &Derived::f );
Function<double (double)>::NonStatic( &Derived::calc );
```

Though we have changed nothing in the implementation of our class, it also can be used for functions in a consistent manner. (Because there is no difference between the implementation of checking functions and attributes, we decided to join classes `Attribute` and `Function` into a `Member` class in our final solution.) The only difference occurs when we're parameterizing our template: we specify function types instead of attribute types. Function qualifiers, such as `const`, naturally fit into this construction:

```
// --- const signature
typedef void Signature(int) const;

struct S {
    Signature f; // --- also can be written as void S::f(int) const
};

// --- Example resulting true
bool result = CONFORMS( Function<Signature>::Member<S>(&S::f) );
```

The type definition may be surprising: keyword `const` is meaningless except for member functions. However, the language standard allows such definitions so as member functions can be defined later, such as `f()` in class `S`. (Note that despite the standard, many compilers do not accept such type definitions). Exploiting possibilities of this feature, we gain a consistent way to check function signatures with modifiers.

Unfortunately this construction leads to compile time errors in some cases. If a function has several overloaded instances, and none of them matches the required signature, the compiler flags the function pointer (e.g. `&S::f`) to be ambiguous. The solution for this limitation needs further work.

3.3 Types

It is often required for a template parameter to define a (nested) type, e.g. a container should have a dependent iterator type. This concept can be implemented using the SFINAE rule[3]. Because the name of the type must not be hardwired in a general solution, we are forced to use macros to solve the problem. One macro is required to ease the definition of the checker functions, the other is to provide readable and comfortable usage.

```
// ----- Macros for easier definition
#define PREPARE_TYPE_CHECKER(NAME) \
template <class T> \
typename enable_if< sizeof(typename T::NAME), Yes >::Result \
check_##NAME(Type2Type<T>); \
\
No check_##NAME(...)

// ----- Macro for easier usage
#define TYPE_IN_CLASS(NAME,TYPE) check_##NAME( Type2Type<TYPE>() )

// --- Definition in global or accessable namespace
PREPARE_TYPE_CHECKER(iterator);

// --- Call check anywhere where variables can be defined
bool result = CONFORMS( TYPE_IN_CLASS(iterator, MyContainer) );
```

[3] A similar solution for this problem was already introduced in [3], but it was usable only for a predefined name and had to be rewritten for each other name.

Class Type2Type is part of the Loki library [4], and is used to differentiate between overloaded function variants without instantiating objects of the parameter type, what may have huge costs and unknown side effects. Type2Type provides a lightweight type holder with a typedef inside and allows argument type deduction the same way as a conventional parameter.

We use class enable_if to provide return type Yes for every conforming case. For the first argument of enable_if, we have to specify a boolean template parameter, hence we use sizeof() to "convert" the inspected type into an integer value, which can be interpreted as a boolean.

Because a checker function for each type name must be declared before it can be used, the PREPARE_TYPE_CHECKER macro must be called in advance with the name to be checked as an argument, at any place in the program where global functions can be defined. After preparation, the check can be made similarly to other checks using the TYPE_IN_CLASS macro. In the last line of the example, we check whether our container class has a nested type with name iterator.

3.4 Member names

Unfortunately all of our previously implemented data member and member function checks were based on the assumption that at least the name of the inspected member exists in the class. Otherwise, a compile time error occurs since the referenced member (e.g. &Derived::var) cannot be found inside the class. Therefore it is essential to check the existence of member names, i.e. the existence of a attribute or function (of any type) with a given name.

The solution for this problem is very similar to the one for inspecting nested types in 3.3. The only difference is in the conversion to a boolean parameter for enable_if. For functions and attributes, we are able to use the address operator&[4] instead of sizeof(). For nested types, sizeof() was our only choice, because pointers cannot be set to types.

Based on the very same principles, we can define our functions to check member names:

```
// ----- Macros for easier definition
#define PREPARE_MEMBER_CHECKER(NAME) \
template <class T> \
typename enable_if< &T::NAME, Yes >::Result \
checkName_##NAME( Type2Type<T> ); \
\
No checkName_##NAME(...)

// ----- Macro for easier usage
#define MEMBER_IN_CLASS(NAME, CLASS) \
    checkName_##NAME( Type2Type<CLASS>() )

// --- Definition in global or accessable namespace
```

[4] Note that in C++, all pointers are accepted as true except for null pointers.

```
PREPARE_MEMBER_CHECKER(size);

// --- Call check anywhere where variables can be defined
bool result = CONFORMS( MEMBER_IN_CLASS(size, MyContainer) );
```

In the last line, we are able to check whether our container class has a function *or* attribute with name `size`.

4 Usage

4.1 Elementary Concepts

Checking atomic concepts is quite user friendly. Still, inspecting dependent names require some macro magic:

```
// --- Defines checker functions
PREPARE_TYPE_CHECKER(iterator);

...

// --- Call checker function later
bool result = CONFORMS( TYPE_IN_CLASS(iterator, MyContainer) );
```

In the first call we prepare our checker function: the macro is expanded into a function definition that searches for a dependent type with name `iterator`. Because we have to define a new checker function for each inspected name, we think this is the most confortable way to provide such definitions. Certainly, this has to be done only once, before making any calls. Later a call to the defined checker can be done. In the example we verify if our container class has an iterator defined. We can inspect dependent member names similarly using macros `PREPARE_MEMBER_CHECKER()` and `MEMBER_IN_CLASS()`.

If we know that such a dependent name exist, remaining inspections are much more natural without many macros. E.g. the search for a comparision operator looks as follows:

```
bool result = CONFORMS(
    Function<bool (const T&, const T&)>::Static(&operator==) );
```

In the above code, `Function<bool (const T&, const T&)>` specifies that we expect to have a function with this signature. We call function `Static()` on this type indicating that we expect to have a global comparision operator. Finally we specify `&operator==` as the inspected function. If the function parameter is exactly the same as we expected, `result` evaluates to true, otherwise it's false. We can verify attribute types similarly.

4.2 Assembling concepts

Elementary concepts are not really useful in themselves. For practical use, we have to combine several elementary conditions to express the actual requirements for a type parameter. In this section we present our approach for assembling our basic conditions into practically used, complex concepts.

Because all of our concept checks result a boolean value, assembly means a simple application of logical operations in our case. In other libraries there was an implicit *and* between listed conditions, what does not necessarily hold for all the cases. In most cases we use *logical and* (`operator &&`), indeed, but other logical operators, such as `operator!` and `operator||` should be supported, too. For example, concept `LessThanComparable` may require that a type must have a member comparision (`T::operator<`) *or* a global comparision operator `::operator<`). Raising an error for check failures, this concept can be expressed as follows[5]:

```
template <class T> struct LessThanComparable
{
   enum { Conforms =
      // --- Check appropriate type for member
      CONFORMS( Function<bool (const T&) >::NonStatic(&T::operator<) )
   ||
      // --- or global operator
      CONFORMS( Function<bool (const T&, const T&)>::Static(&operator<) )
   };
};

// --- Example of usage
template <class Num> struct MyClass
{
    BOOST_STATIC_ASSERT( LessThanComparable<Num>::Conforms );
    ...
}
```

We can see that the introspection code and the usage of the result is clearly separated. In `LessThanComparable`, we define our introspection criterias and calculate the result. In `MyClass`, we define our action utilizing the result, which is raising a compile time error in our case. We could also define any other action, e.g. use compile time adaptation techniques depending on the result.

4.3 Further Improvements

Based on discussed methods, we can check the exact type for members. However, we often do not care about the exact type in practice. Instead, we want

[5] Note that we did not consider the case when name `operator<` does not exist. A check failure leads to compilation error anyway because of the static assertion.

to know if a variable is *usable* (i.e. convertable[6]), which cannot be expressed using the `Attribute` class above. E.g. we would like to require that when we expect a `short`, it can be also a `long` or any class that can be cast to `short`. Our framework should support expression of such non-strict conditions. Note that there is no inheritance relationship between these types; in such cases when there is, even the conventional tests of our previous `Attribute` class result the desired answers. These kind of non-strict expressions should also apply to function signatures, where a `void (long)` signature may conform to a `void (int)` restriction (such functor conversions are possible in `boost::functor`).

We should also get rid of the drawbacks of current implementation, such as compilation error for ambigous operators. This would require a change in the library structure, because this drawback is a direct consequence of referencing members by name when using them as function arguments.

5 Summary

Template introspection would have serious advantages compared to previous solutions, like `requires` macros of g++, the concept library of boost [9] or static interfaces [1].

Boost concept checks forced the instantiation of template features by explicit calls to required features in predefined concept classes, which yields compile time errors in cases of missing features. Thus it is impossible to use concept classes in cooperation with template metaprogramming tools, e.g. `boost::mpl`. It forces the user to partially duplicate functionalities of `boost::concept` and possibly leads to parallel development efforts. Separating execution of actions based on introspection results a more orthogonal design. In this case, the concept library consists of of widely reusable elementary plus predefined complex conditions and libraries (like `boost::mpl`) are able to utilize introspection results. Naturally, one action based on introspection results can be abortion of compilation using `BOOST_STATIC_ASSERT`.

References

1. Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First Workshop on C++ Template Metaprogramming, October 2000
2. J. Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First Workshop on C++ Template Metaprogramming, October 2000
3. David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2002)
4. Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)

[6] Note that this is a different problem than the one has been solved with macro SUPERSUBCLASS in [4].

5. Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
6. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
7. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)
8. Jaakko Jarvi, Jeremiah Willcock, Andrew Lumsdaine: Concept-Controlled Polymorphism. In proceedings of GPCE 2003, LNCS 2830, pp. 228-244.
9. The Boost concept checking library.
   `http://www.boost.org/libs/concept_check/concept_check.htm`
10. The Boost metaprogramming library.
   `http://boost.org/libs/mpl/doc/index.html`