Towards a general template introspection library

István Zólyomi and Zoltán Porkoláb

Department of Computer Science, 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. Unlike other languages (Ada, Eiffel, etc), C++ does not directly support checking requirements on template parameters (i.e. concept checking). However, many articles introduce ad hoc solutions based on special language features. In this paper we propose a structure for a general introspection library which supports easy expression and combination of basic orthogonal requirements, providing the possibility to avoid reimplementation of simple checks for every similar concept. Based on these building blocks, it is possible to express highly complex constraints on template parameters, contrary to languages having builtin support for a limited set of constraints only. Our library enables a checking method that takes the advantages of previous solutions, such as REQUIRE-like macros and static interfaces. Our implementation is non-intrusive, relies only on standard C++ language features and results no runtime overhead.

1 Introduction

Generative programming is a rapidly developing and expanding area in computer science of our days. It includes generic programming, what aims to create general and reusable components, and template metaprogramming, what tries to replace calculations from the program (run time) to the compiler (compile time). In C++ these paradigms are based on the template facility of the language, which provides parametric polymorphism. Templates 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 behaviour 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.

Let us clarify this with an example: using a class `T` (that has no comparison `operator<`) as type parameter for a given `C` container template, `C<T>` is legal even if `C` defines a `sort()` operation, which depends on comparison of objects of type `T`. Because of the lazy evaluation strategy, the compiler does not try to instantiate

the sort operation, therefore does not detect the lack of `operator<` in `T`. Later, when a client happens to call the `sort()` function and the function is about to be instantiated, the compiler flags the lack of `operator<` in `T` and raises an error. This holds even for misspelled functions, e.g. a call `srot()` instead of `sort()` in a function template body would not be recognized until an instantiation attempt. Thus the compiler may accept code expected to be refused and may raise errors later unexpectedly.

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]). 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.

According to Stroustrup, the flexibility of this solution makes concepts unnecessary. He considered concept checking to be even harmful. Contrary, there were heavy efforts to implement concept checking on template parameters since the introduction of templates in C++. In many cases, instantiation of templates can be explicitly forced, thus drawbacks of lazy instantiation can be avoided.

First ad hoc solutions were based on forcing instantiation of required template features manually, e.g. in constructors. Later, an improved version of this technique by Siek [2] was able to avoid runtime overhead for such instantiations. This solution is currently used in several libraries, e.g. in the STL implementation of g++. 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, similarly to interfaces in e.g. Java.

It would be very important and profitable to have a standard, well designed concept checking library: it 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 [10]) 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 more sophisticated programming techniques, such as compile time adaptation. 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. Based on compile time boolean results, we also would be able to express relationships between concepts, using arbitrary logical operations, while previous libraries used an implicit *and* connection between all conditions. For example, a type can be serialized to `cout` using `operator<<` *or* has member function `print()`. Another example can be a type having *no* public constructor (e.g.

singletons). Using either builtin logical operators or custom metaprogramming calculations, we are even able to exploit the lazy evaluation of logical expressions.

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 they are both have to be written from scratch using current libraries if they are not already implemented).

Thirdly, until now, no discussion was made about what would be useful to be expressed as a concept and what minimal orthogonal set of (meta)operations would be required to cover all possible checks. Relying on an orthogonal and complete set of basic concept implementations, we would be able to create a well designed concept library.

Finally, 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. We cannot be sure if we're all searching for the solution at the wrong place, or it is theoretically impossible to implement such concepts based on features of the current language standard. The limitations of concept checking in C++ is still *terra incognita*.

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. Rather, an elementary action should be provided to interrupt compilation with custom messages.
2. Concept checking should be factorized to orthogonal, elementary conditions. We should give tools for constructing compound concepts.
3. The library should be non-intrusive and extensible.

In this paper we aim to provide a general and comprehensive, non-intrusive framework for expressing basic concepts in C++. Our checks result compile time constants instead of compile errors, thus allowing compile time adaptation 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 Programming techniques

Throughout this paper we use several special language features and techniques. In this part we give an introduction to our implementation "tricks".

Firstly, we have two distinct classes `Yes` and `No` marking true and false values by their different size[1]. These classes will be return types of functions, thus marking which variety of overloaded functions is actually chosen for the current function arguments.

```
typedef char No;                          // --- Type meaning false
typedef struct { char dummy[2]; } Yes;  // --- Type meaning true
```

The size of `Yes` is clearly more than the size of `No`, therefore using the `sizeof` operator we can distinguish these classes during compilation time. Instead of manual calls of operator `sizeof`, it is easier to read the code that uses the following class and macro:

```
// --- Type to return result as a bool constant
template <int> struct Conforms;

template <> struct Conforms< sizeof(No) >
    { enum { Result = 0 }; };

template <> struct Conforms< sizeof(Yes) >
    { enum { Result = 1 }; };

// --- Provide more readable form of checking result
#define CONFORMS(PARAM) Conforms< sizeof(PARAM) >::Result
```

Class `Conforms` uses template specialization what allows us to use different bodies for templates with different parameters. `Conforms` does not have an implementation for the general case, providing that any parameter other than `sizeof(No)` or `sizeof(Yes)` leads to compilation error. For an actual argument `sizeof(No)`, the class holds `0` as an enumerated value (being equivalent to a `bool` with a `false` value), otherwise it holds `1` for `sizeof(Yes)` (same as `true`). Finally we define macro `CONFORMS` to give a shortcut for the result.

Another common technique is using ellipses (marked as "`...`" in function signatures). Ellipses mean "match all number and type of parameters". It enables creation of default rescue branches while overloading functions: when the match for all other signatures fail, it surely succeeds. Because ellipse have a minimal priority, it is matched only if no other signature matches the actual arguments. Though ellipses are often considered to be dangerous, we do not actually try to access the parameters of such functions. Ellipses are used to decide whether the actual parameters match the signature of another function or not, hence our library remains safe.

```
Yes f(double);                // --- use different return types
No  f(...);                   // --- use ellipse in rescue case

bool a = CONFORMS( f(2) );    // --- calls first, results true
bool b = CONFORMS( f("two") ); // --- calls rescue case, results false
```

[1] For a detailed explanation of this technique, see [4].

Using the `sizeof` operator (in the implementation of macro `CONFORMS`) on the result of the function call, the function itself is no actually executed, because size of the result type can be deducted anyway. Not executing any function body, we have no runtime overhead penalty for checking. Since no implementation for the function is needed, a single declaration is enough, no definition is required.

We also use the so-called SFINAE rule of C++, what is an acronym for "substitution failure is not an error" (see [3]). It is also often referenced as "two phase lookup". This principle is applied when the compiler tries to instantiate a function template, but deduction of template parameters results in a type error. In this case, instead of raising an error, it tries to use other overloaded instances of the same function. If the compiler finally succeeds, the failure is suppressed and is not considered as an error; it will be flagged only if all instantiation attempts have failed.

```
template <class T>
typename T::iterator f(T t) { return t.begin(); }

void f(...) {}     // --- rescue case

f( list<int>() );  // --- calls first
f( 2 );            // --- no int::iterator, calls second
```

Above we defined two functions to check the existence of type `iterator` declared inside a class. Calling function `f()` with a `list<int>` argument, the first variety of the function is used: type `T` is deducted from the type of the argument, and it has an iterator type, thus the signature is valid. This not holds for the second call: `int` has no nested types at all, thus the signature is invalid and instantiation of the template function fails. According to the SFINAE rule, the compiler struggles on to search for other overloaded versions of the function, and finds our rescue function with the ellipse. Therefore the last line calls this rescue function and the instantiation error will be suppressed.

We can exploit the SFINAE rule directly with the definition and use of the following template class[2]:

```
template <bool, class> struct enable_if {};
template <class T> struct enable_if<true,T> { typedef T Result; };
```

Class `enable_if` provides the possibility of checking arbitrary compile time conditions on any type. It defines the second parameter as `Result` inside the class, if the condition (as its first parameter) is true; otherwise it does not contain anything. Type `enable_if` has to be included in the signature of a function, and additionally a rescue function is needed in the following manner:

```
// --- Declare function with enable_if specifying custom condition
template <class T> enable_if<MY_CONDITION, Yes>::Result f(T);

No f(...);                          // --- rescue case
bool result = CONFORMS( f(1) );     // --- verify condition on type int
```

[2] Class `enable_if` is explained in detail in [8], and is part of the Boost library [9].

If `MY_CONDITION` is true, the result type of `enable_if` will be `Yes`, thus the signature of the first function will be legal and preferred. If the condition is false, the signature is illegal, and the rescue case is found. Hence the result type will be `No`. We can use the above defined `CONFORMS` macro to check the result type of the selected function variety.

3 Elementary conditions

In this section we give a general solution for several basic conditions. However, 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. These works provide good and comprehensive libraries, hance we do not intend to reinvent the wheel: we concentrate on concepts that still do not have a comprehensive or general solution. Furthermore, many of our ideas were inspired by these libraries so effective details and solutions are similar at some places.

All of our concepts return the results of the check as a compile time boolean constant, so as we can specialize our templates acccording to the check result, hence exploiting compile time adaptation techniques. Furthermore, using constants we are able to write logic expressions, thus expressing relations between conditions. We do not directly support raising errors for unsuccessful checks. However, if an error is needed in the case of a check failure, a simple compile time assertion for the result can be used instead, such as `STATIC_CHECK(result, MY_ERROR_MESSAGE)`[3].

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

- Constraints on the type, e.g. size, modifiers, etc[4]
- 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 cannot check 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. They are discussed in section 6.

[3] This example is based on the compile time assertion macro of Loki [4]. Boost [9] provides a similar solution for such assertions.

[4] These constraints already have an appropriate solution as presented in [4] and [3], hence we do not discuss them here.

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(...);

    // --- 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[5]. 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 the 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] Global and namespace variables (and later, functions) could also be checked using function `Static()`.

5. The result size is checked and a compile time boolean constant is finally achieved.

For a deeper understanding and becoming more familiar with 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 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[6], as shown in 2. Because the name of the type must not be hardwired in a general solution, we are forced to use macros to solve the

[6] 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.

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) );
```

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&`[7] 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
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 Assembling concepts

In section 3 we have created our basic concepts. For practical use, we have to combine several elementary conditions for expressing the actual requirements of 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, in our case, assembly means a simple combination using logical operations. In other libraries there was an implicit *and* between listed conditions, however it does not hold for all 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:

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

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

```
   ||
      // --- or global operator
      CONFORMS( Function<bool (const T&, const T&)>::Static(&operator<) )
   };
};

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

The example uses the `STATIC_CHECK` macro of Loki which enables custom error messages as its second parameter. In our user class `MyClass`, besides having an assertion on the result (as above), we could also use compile time adaptation techniques for the result.

5 Further work

5.1 Platforms

Our production code was placed into a single header file *concept.h*[8]. To minimize dependencies from other libraries, we did not use any third party code. However, two code fragments were used without modification from other projects is class `Type2Type` from Loki and `enable_if` from boost. To avoid dependencies from these libraries, we placed these very simple class definitions into header *concept.h*.

Altough we used only standard C++ features in our implementation, compilers are still far behind the current language standard. We were able to compile our whole framework on compilers Intel 8.0 and Visual C++ .Net 7.1. Most of the code could be compiled on Gnu C++ 3.3, but failed to implement SFINAE in an appropriate way. Unfortunately, we did not have the possibility to test Comeau C++.

The only compiler that compiled flawless code was Intel 8.0. Visual C++ was not able to correctly interpret `MEMBER_IN_CLASS`, however accepted it syntactically. Features compiled successfully by GNU C++ were interpreted according to the language standard.

We should be able to "port" a working framework to widely used compilers, hence features supported by most compilers should be used whenever possible.

5.2 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

[8] This file can be downloaded from `gsd.web.elte.hu/Publications`.

to know if a variable is *usable* (i.e. convertable[9]), 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`, see [9]).

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.

6 Open questions

Despite checking many kind of concepts has an appropriate solution, there are several questions left open in this area. Perhaps the most important is checking the existence of a public constructor for a class, e.g. a default constructor[10]. Unlike destructors, constructors are not conventional members of the belonging class (e.g. no member pointer can be set to a constructor), therefore they cannot be referenced explicitly as a function. This forbids the use of our above presented method for constructors.

Similarly, we could find no solution for many template introspection issues. Is a function virtual? How many descendents and ancestors does a class have? Is it abstract? Is it a POD type? Does it require memory in heap?

This kind of problems raise the question: what is the limitation of C++ template introspection? We need a strict theoretical work inspecting:

- What is the minimal orthogonal implementation for covering already solved concept checking problems?
- What other concepts are expected to have a solution based on standard C++ language features?
- What are the concepts that are *theoretically impossible* to be solved (if there is any)?

7 Summary

Template introspection has serious advantages compared to previous solutions, like `requires` macros of g++, the more sophisticated concept library of boost [10] or static interfaces [1].

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

[10] Note that boost [10] is able to require its existence, but it raises compile error for a check failure, and negation of this condition (e.g. for singletons) is impossible.

Conventional concept checks forced the instantiation of template features by manual calls to required features, which yielded compile time errors in cases of missing features. Our library does better than that: it provides compile time boolean values as check results. At the same time, this solution had the advantage of being non-intrusive, which also applies to our solution.

Static interfaces provide similar compile time boolean results as our framework does. On the other side, they have the drawback of being intrusive since all conforming concepts have to be explicitly specified, similarly to implementing Java interfaces.

Our solution unites the advantages of previous solutions: we have a non-intrusive introspection method providing boolean results, while we are able to express the very same constraints on our classes as these previous libraries and more. Instead of providing a large set of concrete concept checks, we intended to implement elementary building blocks and construction facilities. This way the user can specify and express his own custom concept conditions that can be used for both compile time adaptation and termination of the compile process.

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)
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 Library. `http://www.boost.org`
10. The Boost concept checking library. `http://www.boost.org/libs/concept_check/concept_check.htm`