Constraining Concepts Overload Sets ISO/IEC JTC1 SC22 WG21 P0782R1

ADAM David Alan Martin[*], Erich Keane[†], Sean R. Spillane[‡]

September 28, 2018

1 Introduction

The central purpose of Concepts is to make generic programming approachable for the average developer. In general it makes great strides towards this end, particularly in the capacity of invoking a generic function.

Concepts can be viewed from two directions. One way is to think of them as a mechanism to constrain the *derivation* of types in template instantiation. While languages such as Haskell benefit from this point of view, it does not work so well for C++. A concept, implemented as a type derivation constraint, would necessitate the use of either "autoboxing" of types, or transparently allowing several types to manipulate the same address simultaeneously. Clearly, this is not workable.

An alternative approach is to consider a Concept as a way to limit the available overload set at the time when a call to a generic function is compiled, regardless of whether that function is a member of a class or not. This approach has the benefit of being "zero cost", since we don't require fancy auto-boxing types, and we don't have to violate laid-down rules of the standard. Another key feature of this approach is that *at no time are Concepts ever part of types*. This means that we don't have to mess up name-mangling, nor do we have to add any bloat. On the downside, it means that Concepts are, by nature, highly ephemeral.

Concepts are meant to be simple and unsurprising to use. However, the current Concepts design does not deliver on this promise in the implementation of a generic function. Perhaps surprisingly to the average developer, this is because the feature does not constrain the overload set of a template-concept function itself.

We believe that Concepts should strongly resemble the callable properties of an *interface*. That is, a Concept specifies a collection of declared function types, and *only* those types can be invoked when the Concept is invoked to contrain an instance. This mental model suggests that Concepts should offer a mechanism to *limit* the set of operations which would be visible from within a constrained function to those which are specified by the Concepts used by the constrained function.

The fact that this is not the case in constrained functions would be quite surprising to the developer. Worse, this oversight cannot be corrected once the feature is standardized since this would entail making silent behavioral changes to *existing* code after the release of Concepts in a standard. In other words, this is our *only chance* to get this right.

2 Simple Motivating Example

Let us illustrate the main problem with a simple example. We assume that a Concept, which we dub `IntegerSerializer`, describes the following interface:

```
1 template< typename Instance >
2 concept IntegerSerializer= requires( Instance instance, int value )
3 {
4   // This function member takes an int to a std::string, such that
5   // the string is a suitable representation of the int.
6   { std::as_const( instance ).serialize( value ) } -> std::string;
7 }
```

[*]adam@recursive.engineer
[†]erich.keane@intel.com
[‡]sean@spillane.us

Now, we can define several instances of this Concept, for various desired formats:

```
class SimpleSerializer
{
  std::string serialize( int value ) const;
};
struct PrecisionSerializer
{
  std::string serialize( double preciseValue ) const; // Should fail to compile, since serial
};
struct LoanInterestSerializer
{
  std::string serialize( int interestBasisPoints ) const;
  std::string serialize( double interestRate ) const; // Should fail to compile, since serial
};
```

Finally, we can attempt to use the concept in some code. It should be noted that this code—written using the terse-syntax style—makes it unclear whether the author knew they were writing a template:

```
1 std::string
2 formatLogarithmicValue( IntegerSerializer &serializer, int integerValue )
3 {
4   return serializer.serialize( std::log( integerValue ) );
5 }
```

The issue is in what kind of object we pass to `formatLogarithmicValue`. If we pass a `SimpleSerializer`, then no surprises happen—the `double` is implicitly converted to `int`. If we pass a `PrecisionSerializer`, then the definition of the concept will pass and an implicit conversion to `int` will happen, which is potentially surprising. However, many programmers are reasonably comfortable with the idea of fundamental type conversions. The most surprising case is that of `LoanInterestSerializer`. `LoanInterestSerializer` provides a `double` and an `int` overload. Although the concept requested a function with a signature that accepts an `int`, the overload which accepts `double` in its signature will be called instead. From the perspective of the compiler, the way it will actually compile the function `formatLogarithmicValue` is as if it were written:

```
1 template< typename IntegerSerializer >
2 std::string
3 formatLogarithmicValue( IntegerSerializer &serializer, int integerValue )
4 {
5 return serializer.serialize( std::log( integerValue ) );
6 }
```

At this point, the invocation of `IntegerSerializer::serializer` will be whatever best matches `decltype( std::log( integerValue ))` , which is the overload with `double` as its parameter. This is likely surprising behavior to the author of `formatLogarithmicValue`, as well as the caller of `formatLogarithmicValue`. Both of these authors would expect that the constraints described by the concept would be obeyed, yet paradoxically the overload which was not the best match for the constraint was actually the overload that was actually invoked in the body of the "constrained" function!

The only way for an author of such a constrained function to avoid this, at present, is to rewrite `formatLogarithmicValue` in such a way as to prevent the incorrect lookup. Unfortunately, this requires a level of C++ expertise regarding name lookup and overload resolution which is at odds with the level of expertise expected of the audience of Concepts, viz. the non-expert programmer. Such a rewrite might appear thus:

```
1 template< typename IS >
2 std::string
3 formatLogarithmicValue( IS &serializer, int integerValue )
4 requires( IntegerSerializer< IS > )
5 {
6   return std::as_const( serializer ).serialize(
7   static_cast< int >( std::log( static_cast< double >( integerValue ) ) ) );
```

```
8 }
```

This does not appear to be code that would be expected of the audience targetted by Concepts. Additionally, although one of the authors of this paper is author of `std::as_const`, this is not the purpose nor audience he had in mind when he proposed it. The primary purpose of the static casts and as-consts in this rewrite are actually dedicated to the selection of the correct overload, not to any specific need to have a value in the form of any specific type.

3 An Example at Scale

Let us now consider a more complete example. First, we define the `Show` Concept. This Concept specifies a single required function:

```
1  namespace ConceptLibrary {
2    template< typename Instance >
3    concept Show = requires( Instance instance, int value ) {
4      { std::as_const( instance ).toString() } -> std::string;
5    }
6  }
```

Listing 1: The `Show` Concept.

Next, we sketch out an `Employee` class, which will implement the `Show` Concept. Note please that the intent of the `fire` function is to *terminate* an `Employee`:

```
1  namespace ConceptLibrary::PayrollLibrary {
2    class Employee {
3      private:
4      std::string name;
5      public:
6      Employee( std::string initialName );
7
8      // This satisfies the Show Concept, and the author of this type
9      // knows that.
10     std::string toString() const;
11
12     // The following functions are friend, to indicate that ADL is
13     // intended.
14     friend bool operator ==
15       ( const Employee &lhs
16       , const Employee &rhs );
17
18     // Fire the specified Employee
19     friend void fire( const Employee &emp );
20
21     // Hire the specified Employee
22     friend hire ( const Employee &emp );
23
24     // Return true only if the specified Employee is hired
25     friend bool worksHere ( const Employee &emp );
26   };
27 }
```

Listing 2: `Employee` implementation of the `Show` Concept.

Now, we will sketch out a simple algorithm library. Note please that there is a `fire` function defined here too. It is clearly intended to "fire" off `Show`able objects:

```
1  namespace ConceptLibrary::AlgorithmLibrary {
2    // This "fires" off a Showable object to be processed.
```

```
3     void fire( const ConceptLibrary::Show &s ) {
4       std::cout << "I am intested in " << s.toString() << std::endl;
5     }
6
7     // This is intended to call the fire function, defined above.
8     void printAll( std::vector< ConceptLibrary::Show > &v ) {
9       for ( auto &&s : v ) fire( s );
10    }
11  }
```

Listing 3: `Algorithm` library that uses `Show`able objects.

We should note that neither function defined above makes it obvious that it is in fact a template generic function. This is because this code makes use of the terse-syntax. It is likely that the author would know, at some level, that they were defining a generic function.

Finally, let us sketch out a driver program that exploits our previously-defined libraries:

```
1   namespace UserProgram {
2     void code() {
3       std::vector< PayrollLibrary::Employee > team;
4       team.emplace_back( "John Doe" );
5       AlgorithmLibrary::printAll( team );
6     }
7   }
```

Listing 4: Example driver program.

It should be quite obvious what each of the authors of the code segments above intended. The author of the `PayrollLibrary` wanted to represent `Employee`s at a company. The author of the `AlgorithmLibrary` wished to afford his users the ability to print printable things and needed to write an internal helper method to better organize his code. The author of `UserProgram` wanted to combine these reusable components for a simple task.

Unfortunately for these developers, a subtle behavior of name lookup in function templates resulted in the termination of employees, rather than the intended call to an implementation detail! This happened because both the `Employee` class and the `AlgorithmLibrary` chose to name a function `fire`. The implementation detail in `AlgorithmLibrary` happened to collide with the name of some API function in the `Employee` object being processed. Recall that the author of `AlgorithmLibrary` is very likely unaware of the fact that types may exist which are `Show`able and yet interfere with the name he chose for his internal implementation detail.

The authors of this paper recognize the importance of respecting the original intent of the programmers of these components without burying them in the details of defensive template writing. These kinds of examples will come up frequently and perniciously in codebases which import third party libraries and work in multiple groups each with different naming conventions. Even without variance in naming conventions, names that have multiple meanings to multiple people are likely to be used across disparate parts of a codebase, and thus they are more likely to exhibit this pathological behavior.

4 Why it is Important to Address this Now

Should the terse syntax be accepted into the current standard, without addressing this issue, then future attempts to repair this oversight in the language specification, leave us with one of two incredibly unpallateable alternatives and one unsatisfying one:

1. Make constraint violations an error, thus requiring extreme verbosity.
2. Silently change the meaning of existing code, with ODR implications, because these constrained functions are templates.
3. Introduce a new syntax for indicating that a function definition should be processed in a manner which is more consistent with average programmer expectations.

In the first case, vast amounts of code will fail to compile in noisy ways. After that point, the user code would need to be rewritten, in a manner similar to the necessary rewrites as described above. In the second case, massive fallout from ODR, silent subtle semantic changes, and other unforseen dangers lie in wait. In the third

case, the benefits of the "natural" syntax are lost, as the best syntax for beginners is no longer the natural syntax! This obviously defeats the intended purpose of Concepts with a natural syntax.

5 Some Design Philosophy

There are other cases where current Concepts can fail to prevent an incorrect selection of operation. This fails to deliver upon a big part of the expected benefits of this language feature. The comparison has been drawn between C++ Virtual Functions and Concepts. As Concepts are being presented to bring generic programming to the masses, it is vital that 3 core safety requirements be considered. These requirements are similar to aspects of Object Oriented Programming:

1. An object passed to a function must meet the qualifications that a concept describes. This is analagous to how a function taking a pointer or reference to a class has the parameter checked for substitutiablility. The current Concepts proposal provides this extremely well.

2. An object written to be used as a model of a concept should have its definition checked for completeness by the compiler. This is analagous to how a class is checked for abstractness vs concreteness. The current Concepts proposal lacks this. However, this is approximated very well by the concept checking machinery. This guarantees that every class which is matched to concept provides a definition for every required operation under that concept, thus satisfying the requirements of the concept.

3. A constrained function is only capable of calling the functions on its parameters that are described by its constraining Concepts. This is analagous to how a function taking a pointer to base is only allowed to call members of the base -- new APIs added in any derived class are not considered to be better matches, ever. The current Concepts proposal lacks anything resembling this, and this oversight has yet to be addressed. It is this deficiency which our paper seeks to remedy.

We propose that the Concepts feature is incomplete without constrained overload set for usage, thus satisfying the third requirement of any interface-like abstraction. It is vital that we explore this issue. We recognize that some complexity in the space of constrained generics will always be present, but we feel that it is best to offload this complexity to the author of a concept rather than to the implementor of a constrained function. This is because we believe that fewer concept authors will exist than concept "users".

Additionally, the level of expertise of a concept author is inherently higher than the intended audience of constrained functions. In the worst case scenario, a naive definition of a concept will merely result in a few missed opportunities for more suitable overloads to handle move semantics, avoid conversions, and other shenanigans

6 What This Paper is *NOT* Proposing

1. Any requirement on optimizers to make any aspect of the code generated by this solution more efficient
2. Definition Checking
3. C++0x Concept Checking
4. C++0x Concept Maps
5. The generation of "invisible" proxy types
6. The generation of "invisible" inline proxy functions
7. Dynamic dispatch
8. Function call tables
9. Implicit generation of adaptors
10. Any form of extra code generation
11. Any form of extra type generation

7 Our Proposed Solution

We propose a moderate alteration of the overload resolution rules and name lookup rules that statically filters out some overloads based upon whether those functions are used to satisfy the concept's requirements. This constrained overload set hides some functions from visibility in the definition of constrained functions. No change should be necessary to the rules of name lookup itself; however, our new overload resolution rules will affect the results overload resolution on unqualified name lookup in constrained functions-- this is by design. We also suggest that it might be desirable to borrow a keyword (we suggest `explicit`) to indicate that this amended lookup rule should be followed. The need for this keyword to enable these lookup rules will be discussed further in the "Design Considerations" section of this paper. Specifically, our design is to change overload resolution to be the following (taken from cppreference.com's description of the overload resolution rules):

Given the set of candidate functions, constructed as described above, the next step of overload resolution is examining arguments and parameters to reduce the set to the set of viable functions To be included in the set of viable functions, the candidate function must satisfy the following:

Current overload resolution	Desired overload resolution
If there are M arguments, the candidate function that has exactly M parameters is viable	
If the candidate function has less than M parameters, but has an ellipsis parameter, it is viable.	
If the candidate function has more than M parameters and the $M+1$'st parameter and all parameters that follow must have default arguments, it is viable. For the rest of overload resolution, the parameter list is truncated at M.	
If the function has an associated constraint, it must be satisfied (since C++20)	
	If at least one of the arguments to the function is constrained and that function was found by unqualified name lookup and the lookup found a name that is otherwise not visible at the calling location, then the function is only viable if that function was necessary to satisfy the argument's concept constraint. (this paper)
For every argument there must be at least one implicit conversion sequence that converts it to the corresponding parameter.	
If any parameter has reference type, reference binding is accounted for at this step: if an rvalue argument corresponds to non-const lvalue reference parameter or an lvalue argument corresponds to rvalue reference parameter, the function is not viable.	

8 Objections, Questions, and Concerns

8.1 Isn't this just C++0x Concepts with definition checking all over again?

No. C++0x Concepts used mechanisms and techniques which are drastically different to the solution we have proposed. In the original C++0x Concepts, the compiler was required to create invisible wrapping types (Concept Maps) and create inline shimming functions, and to calculate the set of archetype types for a Concept. C++0x Concepts were difficult to inline and relied upon the optimizer to eliminate the overhead imposed by these automatically generated constructs. C++0x concepts also provided "definition checking", which was desirable in that a template function could be checked for correctness before instantiation. C++0x concepts used the "Concept Map" and archetype computations to decide whether a template would be correct. This compiletime computation of an

archetype runs into a manifestation of the halting problem, which makes generalizing this solution to user defined concepts a problematic proposition at best.
By contrast, our solution does not involve the generation of such machinery. We require no generation of any adaptors, maps, or proxies. Instead, we propose altering and refining the lookup rules to further obey the restrictions imposed by Concepts in a manner similar to what is already in the existing design. We feel that this is appropriate, because Concepts already requires some alteration to the lookup rules, and our design appears to be consistent with the general lookup rule restrictions thereby imposed. Concept restrictions, in their current form, are enforced in C++ through lookup rules, not through any other mechanism. Our solution merely adds more rules to the set of lookup rules employed in concept processing. This also means that our solution is not capable of providing definition checking -- the lookup rules that this solution alters are those which occur during the second phase of name lookup, after template instantiation.

8.2 Will I be able to call internal helper functions to my constrained function using an unqualified name?

Yes. We place no restrictions on the calling of functions in namespaces that are unrelated to the concept used in a constraint. The namespaces associated with the types that are constrained are also still searched, but only the names which are necessary (in some fashion) to meet the requirements of the concept are considered to be viable. In some sense this is the existing Concepts restriction on calling a constrained function applied in reverse -- constraints restrict which functions are called based upon their arguments. The current restriction prevents calling a function which is not prepared to accept a type. Our refinement prevents calling a function which is not presented as part of the requirements on a type. This example is illustrated in our "at scale" example, and it is one of our primary motivations.

8.3 Isn't your real problem with {ADL, const vs. non-const overloads, overload resolution, dependent lookup, etc.} and not with the lookup rules of Concepts today?

Absolutely not. We have examples of unexpected selection of operation each and every one of these cases. We are not convinced that our problem is with every single one of the above aspects of the language. There are some cases which will be redundantly resolved by improving those aspects of the language; however, many problem cases within each of these domains still remain. This is especially true of ADL functions. ADL functions are intended to be part of the interface of a class; however, a constrained value is also a constrained interface.

8.4 How do I actually invoke ADL functions that I want invoked in my constrained functions?

ADL functions on an object which are actually part of the interface defined by the concept that constrains the calling function are still preserved as part of the viable overload set. This makes them likely to be the best match for an unqualified call, unless a more specific overload is available in some other reachable namespace. If you wish to call ADL functions which are not part of the constraint, then one always has two viable options. The first is to use a direct "using std::foobar" approach, and the second is to adjust the definition of a Concept to include this ADL operation.

8.5 What about calling efficient 'swap' on an 'Assignable'?

This is actually a special case of the above concern. In this case, there are at least two viable options. The first is to add `swap( a, b )` to the requirements of the `Assignable` concept. The second is to make `std::swap` have an overload which accepts a value which is a model of the `Swapable` concept. An unqualified call to `swap` after the traditional `using std::swap;` declaration will invoke that `Swapable` overload, thus giving the correct behavior. In addition, this has the added benefit of making any direct call to `std::swap` in any context always take the best overload! Although this library change is not proposed by this paper, the authors would strongly support such a change.

8.6 Is this going to give me better error messages?

Although this is highly dependent upon the details of an implementation, it is possible that better error messages would be possible under this proposal. The instantiation of a template which requires some specific operation which

is not part of a concept should give a better error message---something along the lines of "Function not found during constraint checking."

8.7 What about implicit conversions needed in ?

Because the function selected by overload resolution must be part of the operations necessary to satisfy the constraints of the specified concept, all implicit conversions which are necessary to invoke that function are also part of the operations necessary to satisfy that concept. This means that any set of implicit conversions provided by a type which are necessary to invoke a selected overload should be "whitelisted" for use in constrained functions when calling that overload.

8.8 Will I be able to invoke arbitrary operations on my constrained parameters which are not part of the concept?

Any function which is available directly in the namespace (directly or via a `using` statement) of a constrained function may be called. Any function which is a template but is not constrained will be called as an unconstrained function. In that context of an unconstrained template, any function may be invoked as normal. We also propose that an intrinsic cast-like operation could be added which will revert a constrained variable to an unconstrained variable, to permit calling functions in an unconstrained fashion for various purposes.

9 Design Considerations

Any design that proposes to change lookup rules should not invalidate code written under those rules today. Because of this, we do not propose that the lookup rules should be changed when evaluating names within a "classical" template context. However, we see a number of opportunities to apply our modified lookup scheme:

9.1 Terse syntax constrained functions

The terse syntax intends to open generic programming to a wider audience, as discussed earlier. We feel that it is obvious that such terse syntax functions be subject to rules which provide a more intuitive result. Therefore we suggest that any terse syntax considered by the committee must havethe intuitive semantics provided by our proposal.

9.2 Template syntax constrained function

9.2.1 All constrained function templates

Any expansion of a terse syntax from the terse form into a "canonical" production of a constrained template function declaration could automatically have these rules applied. This seems fairly obvious in many respects, because the purpose of Concepts is to afford better selection of applicable functions in name lookup and overload resolution. When a user writes a constrained function, even using template syntax, he or she is explicitly choosing to have the semantics of Concepts applied to their function. Therefore, it seems a reasonable choice to make every constrained function obey these lookup rules.

9.2.2 Opt-in for these rules as part of the definition of a constrained function template

A user trying to modernize a code base by adding constraints to existing template functions, may wind up causing subtle changes in the semantics and or ODR violations. Additionally, the template expert is already intimately familiar with the consequences of C++'s uninituitive lookup rules in templates and may wish to leverage the semantics afforded by these rules in the implementation of his template function—he only wishes to constrain the callers, but not himself. Therefore it may be necessary to control the application of this modified rule through the use of a signifying keyword. We propose `explicit template< ... >` as this syntax, as it reads reasonably well and clearly indicates intent. Although this proposed syntax uses the `template` keyword, which is already indicitive of potential lookup dangers, it eschews the pitfalls for the `template< ... >` case.

Regardless of whether the new lookup rules are opt-in or opt-out, the language loses no expressivity. It is possible to choose the opposite alternative through other syntax.

9.3 Behavior of These Rules Under Short Circuit of Disjunction

We preserve the viability of overloads which are found on either branch of a disjunction, because a user would reasonably expect these overloads to be available if those constraints are satisfied. For branches of a disjunction which are not satisfied, those overloads will be unavailable, as the constraint wasn't satisfied. This seems to result in a viable overload set which most closely conforms to user expectations. The overall compile-time cost of this added checking should be proportional to the overall cost of treating a disjunction as a conjunction for this feature.

10 Conclusion

In P0726R0 the authors asked if Concepts improve upon the expressivity of C++17. The response from EWG was mixed. The fact is that, although Concepts are more approachable and readable than C++17 `std::enable_if` predicates, they do not provide any new expressivity to the language, nor do they provide any facility for making templates actually easier to write. We feel that this proposal's adoption will give a new dimension to the Concepts feature which will enable the simple expression of C++ generic code in a much safer and more readable style than C++17 or the current Concepts design.

11 Revision History

P0782R0 A Case for Simplifying/Improving Natural Syntax Concepts (Erich Keane, A.D.A. Martin, Allan Deutsch)
The original draft explained the general problem with a simple example. The motivation was to help attain a better consensus for "Natural Syntax" Concepts. It was presented in the Monday Evening session of EWG at Jacksonville, on 2018-03-12. The guidance from the group was strongly positive:

S.F.	F.	N.	A.	S.A.
10	21	22	7	1

12 Acknowledgements

The authors would like to thank Allan Deutsch, Hal Finkel, Lisa Lippincott, Gabriel Dos Reis, Herb Sutter, Faisal Vali, and numerous others for their research, support, input, review, and guidance throughout the lifetime of this proposal. Without their assistance this would not have been possible.

13 References

P0726R0 "Does Concepts Improve on C++17?"