Document Number: P4174R0
Date: 2026-04-2
Title: Named, composable type sets for concept constraints
Author: Emanuel Spiridon (spiridonemanuel23@gmail.com)
Audience: LEWG

Abstract

This paper explores the possibility of adding a library addition to <type_traits> to assist the programmer when writing concept constraints. Concept constraints can be written by using the std::is_same_v tool, however the compile time logic is hard to understand, and at times, are error-prone at scale.

An issue that modern concept constraints have is that, despite how simple it is once you implement fold expressions, they have no deduplication logic, concepts aren't first-class types, and errors will dump every type.

Impact on standard

This library addition integrates seamlessly into <type_traits>, it adds no runtime overhead and has no ABI impact. This library works today using GCC, Clang, or MSVC.

Prior art

Boost.Mp11 provides mp_contains<L, T> and mp_unique<L> which implement the same mechanics under the hood as this proposal. Mp11 operates on any variadic template, unlike this proposal which operates on its own type list, and is a mature and widely used library.

However, Mp11 is a general-purpose template meta programming toolkit, providing no concept integration, no requires-clause idiom and no method to merge designed for the type categorization use case. This library does not address the problem of named, reusable, and mergeable type lists as a user-facing vocabulary for concept authoring.This proposal standardizes the idiom that Mp11 makes possible.

Brigand similarly provides type list primitives with membership testing. Designed as a faster Boost.MPL, and sharing the same limitations, Brigand is a TMP infrastructure, not a vocabulary meant for concept authoring.

There isn't a standard library tool that addresses this issue, the closest approximation is a fold-expanded constraint using any_of:

```
template<typename T, typename… Args>
concept any_of = (std::is_same_v<T, Args> || …);
```

This works well for flat single-site constraints, and is recommended for that use case. It does not provide deduplication across merged type families and does not produce a first-class type that can be passed as a template argument.

Design decisions

Immutability

Lists are immutable once defined, they are meant to be used as either concept constraints or building blocks for a larger list which will then be used as a concept constraint. Either way, the end result is concept constraints, which are compile time logic meaning there isn't much space for dynamic type list allocation.

Interaction with standard concepts

Type lists are designed to be used as concepts, using `contains<L, T>`, which is a concept itself, to check if a type is in a list, meaning that it works with any other concept constraint. A requires-clause that requires a type from an already defined list, in addition with a few filler types, can easily be written without needing to create a whole new list or concept.

Fold-expanded constraints vs type lists

As mentioned earlier, fold-expanded constraints are simple to write, they are very good for simple flat type lists, which I recommend in this case. However, this paper proposes type lists that are able to merge under a new type list, while cutting compile time checking for types mentioned more than once at definition and cutting the duplications out.

This paper proposes type lists meant for large scale concept constraints.

Subsumption

This paper proposes a library approach of type lists, meaning that subsumption is not feasible. Subsumption chooses the most specific overload if the more specific concept is in the creation of the less specific one. So subsumption works only if the programmer creates less specific concepts out of more specific ones instead of creating new less specific type lists.

The approach this paper proposes lets the user choose between creating concepts from other concepts, or if they merge type lists and types to eventually create a blank slate concept.

Without explicit concept hierarchies, the following produces an ambiguity error rather than selecting the more specific overload:

```
using natural_numbers = tag::merge_tags_and_types_t
   unsigned char, unsigned short, unsigned int,
   unsigned long, unsigned long long>;

using numbers = tag::merge_tags_and_types_t
   natural_numbers, float, double, long double>;

template<typename T>
concept is_natural = tag::contains<natural_numbers, T>;

template<typename T>
concept is_number = tag::contains<numbers, T>;

void foo(is_natural auto);  // intended: more specific
void foo(is_number auto);   // intended: less specific

foo(1u);  // error: ambiguous — compiler cannot determine
          // that is_natural is more restrictive than is_number
```

The compiler sees two unrelated container<>::value instantiations with different list arguments and has no mechanism to compare their contents. Subsumption can be achieved manually by defining concepts in terms of each other:

```
template<typename T>
concept is_number = is_natural<T> || is_real<T>;

foo(1u);  // now unambiguous — is_natural subsumes is_number
```

This places the responsibility on the programmer to express concept relationships explicitly, which is consistent with how subsumption works throughout the rest of the standard.

Compile time complexity

This paper proposes a simple approach, meaning that the time complexity for every contains<L, T> check, the time complexity is of O(n), while for every time deduplication is involved, the time complexity is of O(n^2) at type list definition.

Proposed wording

To be provided after LEWG design review.

Proposed additions to <type_traits>:

- std::type_tag_list<Args...>
- std::merge_tags_and_types_t<Args...>
- std::tag_contains<List, T> (concept)

Future directions

This paper leaves place for a potential compiler implementation of type lists that could track the ancestry of type lists for subsumption, additionally, it could implement compiler native lookup which could potentially result in a time complexity of O(1).

Acknowledgements

I thank Jens Maurer and JJ Marr for sending feedback and for their valuable feedback on the design.

Appendix

Compiler compatibility: Clang 15+, GCC 12+, MSVC 19.3+
Github implementation repository