Ordering of constraints involving fold expressions

Document #:	P2963R0
Date:	2023-09-15
Programming Language C++	
Audience:	EWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

Fold expressions, which syntactically look deceptively like conjunctions/subjections for the purpose of constraint ordering are in fact atomic constraints We propose rules for the normalization and ordering of fold expressions over `&&` and `||`.

Motivation

This paper is an offshoot of P2841R0 [1] which described the issue with lack of subsumption for fold expressions. This was first observed in a Concept TS issue.

This question comes up ever so often on online boards and various chats.

- [StackOverflow] How are fold expressions used in the partial ordering of constraints?
- [StackOverflow] How to implement the generalized form of std::same_as?

In Urbana, core observed "We can't constrain variadic templates without fold-expressions" and almost folded (!) fold expressions into the concept TS. The expectation that these features should interoperate well then appear long-standing.

Subsumption and fold expressions over `&&` and `||`

Consider:

```
template <class T> concept bool A = std::is_move_constructible_v<T>;
template <class T> concept bool B = std::is_copy_constructible_v<T>;
template <class T> concept bool C = A<T> && B<T>;

template <class... T>
requires (A<T> && ...)
void g(T...);

template <class... T>
requires (C<T> && ...)
void g(T...);
```

We want to apply the subsumption rule to the normalized form of the requires clause (and its arguments). As of C++23, the above `g` is ambiguous.

This is useful when dealing with algebraic-type classes. Consider a concept constraining a (simplified) environment implementation via a type-indexed `std::tuple`. (In real code, the environment is a type-tag indexed map.)

```
template <typename X, typename... T>
concept environment_of = (... && requires (X& x) { { get<T>(x) } -> std::same_as<T&>; } );

auto f(sender auto&& s, environment_of<std::stop_token> auto env); // uses std::allocator
auto f(sender auto&& s, environment_of<std::stop_token, std::pmr::allocator> auto env); //
    uses given allocator
```

Without the subsumption fixes to fold expressions, the above two overloads conflict, even though they should by rights be partially ordered.

Impact on the standard

This change makes ambiguous overload valid and should not break existing valid code.

Implementabiliy

This was partially implemented in Clang. Importantly, we know that what we propose does not affect compilers' ability to partially order functions by constraints without instantiating them, nor does it affect the caching of subsumption, which is important to minimize the cost of concepts on compile time.

What this paper is not

When the pattern of the `fold-expressions` is a 'concept' template parameter, this paper does not apply. In that case, we need different rules which are covered in P2841R0 [1] along with the rest of the "concept template parameter" feature (specifically, for concepts patterns we need to decompose each concepts into its constituent atomic constraints and produce a fully decomposed sequence of conjunction/disjunction)

Design and wording strategy

To simplify the wording, we first normalize fold expressions to extract the non-pack expression of binary folds into its own normalized form, and transform `(... && A)` into `(A && ...)` as they are semantically identical for the purpose of subsumption. We then are left with either `(A && ...)` or `(A || ...)`, and for packs of the same size, the rules of subsumptions are the same as for that of atomic constraints.

Wording

� Constraint normalization [temp.constr.normal]

The *normal form* of an *expression* `E` is a constraint[temp.constr.constr] that is defined as follows:

- The normal form of an expression `( E )` is the normal form of `E`.
- The normal form of an expression `E1 || E2` is the disjunction[temp.constr.op] of the normal forms of `E1` and `E2`.
- The normal form of an expression `E1 && E2` is the conjunction of the normal forms of `E1` and `E2`.
- The normal form of a concept-id `C<`A_1`, `A_2`, ..., `A_n`>` is the normal form of the *constraint-expression* of `C`, after substituting A_1, A_2, ..., A_n for `C`'s respective template parameters in the parameter mappings in each atomic constraint. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required. [*Example:*

```
template<typename T> concept A = T::value || true;
template<typename U> concept B = A<U*>;
template<typename V> concept C = B<V&>;
```

  Normalization of `B`'s *constraint-expression* is valid and results in `T::value` (with the mapping `T` $\mapsto$ `U*`) $\vee$ `true` (with an empty mapping), despite the expression `T::value` being ill-formed for a pointer type `T`. Normalization of `C`'s *constraint-expression* results in the program being ill-formed, because it would form the invalid type `V&*` in the parameter mapping. — *end example*]
- The normal form of an expression `( ... `*`fold-operator`*` E )` is `( E `*`fold-operator`*`...)`.
- The normal form of an expression `( E && ... && Pack )` or `( Pack && ... && E )` where `Pack` is an unexpanded pack is the conjunction of the normal forms of `(Pack &&...)` and `E`.
- The normal form of an expression `( E || ... || Pack )` or `( Pack || ... || E )` where `Pack` is an unexpanded pack is the disjunction of the normal forms of `(Pack || ...)` and `E`.
- The normal form of any other expression `E` is the atomic constraint whose expression is `E` and whose parameter mapping is the identity mapping.

� Partial ordering by constraints [temp.constr.order]

A constraint P *subsumes* a constraint Q if and only if, for every disjunctive clause P_i in the disjunctive normal form of P, P_i subsumes every conjunctive clause Q_j in the conjunctive normal form of Q, where

- a disjunctive clause P_i subsumes a conjunctive clause Q_j if and only if there exists an atomic constraint P_{ia} in P_i for which there exists an atomic constraint Q_{jb} in Q_j such that P_{ia} subsumes Q_{jb}, and

- an atomic constraint A subsumes another atomic constraint B if ~~and only if A and B are identical using the rules described in [temp.constr.atomic].~~ :

- A is a *fold-expression* of the form $(P\ \texttt{\&\&...})$, B is a *fold-expression* of the form $(Q\ \texttt{\&\&...})$ or $(Q\ \texttt{||}\ \texttt{...})$ and let P' be the *template-argument* corresponding to P in the parameter mapping of A, and let Q' be the *template-argument* corresponding to Q in the parameter mapping of B, $\texttt{sizeof...}(Q')\ \texttt{==}\ \texttt{sizeof...}(Q')$ is `true` P subsumes Q.

- A is a *fold-expression* of the form $(P\ \texttt{||...})$ or $(P\ \texttt{\&\&...})$ and B is a *fold-expression* of the form $(Q\ \texttt{||...})$ and let P' be the *template-argument* corresponding to P in the parameter mapping of A, and let Q' be the *template-argument* corresponding to Q in the parameter mapping of B, $\texttt{sizeof...}(Q')\ \texttt{==}\ \texttt{sizeof...}(Q')$ is `true`, P subsumes Q.

- A and B are identical using the rules described in [temp.constr.atomic].

[*Example:* Let A and B be atomic constraints [temp.constr.atomic]. The constraint $A \wedge B$ subsumes A, but A does not subsume $A \wedge B$. The constraint A subsumes $A \vee B$, but $A \vee B$ does not subsume A. Also note that every constraint subsumes itself. — *end example*]

Acknowledgments

Thanks to Robert Haberlach for creating the original Concept TS issue.

References

[1] Corentin Jabot and Gašper Ažman. P2841R0: Concept template parameters. https://wg21.link/p2841r0, 5 2023.

[N4958] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4958