Ordering of constraints involving fold expressions

Document #:	P2963R3
Date:	2024-06-28
Programming Language C++	
Audience:	EWG, CWG

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 `||`.

Revisions

R3

- Wording improvements following CWG review in St Louis.
- Protect from empty packs inconsistencies by requiring that both constraints involve equivalent template parameters (which ensure they have the same size)
- Add an Annex C entry, as the decomposing of the fold expanded constraint might cause some atomic constraint to have non-bool types.

R2

- Wording improvements following CWG review in Tokyio. Notably, we added a description of how satisfaction is established.
- Clarify that subsumption checking short-circuits. Add a design discussion.
- A fold expression constraint can now only subsume another if they both have the same fold operator. This avoid inconsistent subsumption and checking results in the presence of empty packs.

R1

- Wording improvements: The previous version of this paper incorrectly looked at the size of the packs involved in the fold expressions (as it forced partial ordering to look at template arguments). The current design does not look at the template argument/parameter mapping to establish subsumption of fold expressions.

- A complete implementation of this proposal is available on Compiler Explorer. The implementation section was expanded.
- Add an additional example.

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 A = std::is_move_constructible_v<T>;
template <class T> concept B = std::is_copy_constructible_v<T>;
template <class T> concept 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 be partially ordered.

A similar example courtesy of Barry Revzin:

```
template <std::ranges::bidirectional_range R> void f(R&&); // #1
template <std::ranges::random_access_range R> void f(R&&); // #2

template <std::ranges::bidirectional_range... R> void g(R&&...); // #3
template <std::ranges::random_access_range... R> void g(R&&...); // #4
```

C++23	This Paper

`f(std::vector{1, 2, 3}); // Ok`
`g(std::vector{1, 2, 3}); // Error: call to 'g' is ambiguous`
	`f(std::vector{1, 2, 3}); // Ok, calls #2`
	`g(std::vector{1, 2, 3}); // Ok, calls #4`

[Compiler Explorer Demo]

Impact on the standard

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

Implementabiliy

This was implemented in Clang. Importantly, 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: The template arguments of the constraint expressions do not need to be observed to establish subsumption.

An implementation does need to track whether an atomic constraint that contains an unexpanded pack was originally part of a and/or fold expression to properly implement the subsumption rules (`&&` subsumes `||` & `&&` and `||` subsumes `||`).

Subsection with mixed fold operators

The Kona example

Consider this example provided by Hubert Tong

```
template <typename ...T> struct Tuple { };
template <typename T> concept P = true;
```

```
template <typename T, typename U, typename V, typename X> struct A;


template <typename ...T, typename ...U, typename V, typename X>
requires P<X> || ((P<V> || P<U>) || ...)         // #A
void foo(A<Tuple<T ...>, Tuple<U ...>, V, X> *); // #1

template <typename ...T, typename ...U, typename V, typename X>
requires P<X> || ((P<V> && P<T>) && ...)         // #B
void foo(A<Tuple<T ...>, Tuple<U ...>, V, X> *); // #2


void bar(A<Tuple<int>, Tuple<>, int, int> *p) { foo(p); }
```

In this example, under the rule proposed in R1, of this paper, #A subsumed #B, and so #1 would have been be a better choice. However here, `U` is empty. So A's constraints are equivalent to just `P<X>` which make B more constrained.

The Saint Louis Example

```
struct Magic;
template <typename ...V> struct A;
template <typename, auto V> constexpr auto VTmpl = V;
template <auto V> constexpr auto VTmpl<Magic, V> = !V;
template <typename T> concept CF = VTmpl<T, false>;
template <typename T> concept CT = VTmpl<T, true>;

template <typename ...T, typename ...U, typename V>
struct X *f(A<V, T ...> *, A<V, U ...> *) requires ((CF<V> || CT<T>) && ...);

template <typename ...T, typename ...U, typename V>
struct Y *f(A<V, T ...> *, A<V, U ...> *) requires ((CF<V> && CF<U>) && ...);

void g(A<void, int> *a1p, A<void> *a2p, A<void, Magic> *amp) {
        Y *p;
        p = f(a1p, a2p);
        p = f(amp, a2p);
}
```

In this example both fold operators are the same but because `U` is empty, the second overload's constraints are not satisfied even though subsumption finds it more specialized.

In a previous version of this paper, we disallowed mixed-operators subsumption. However this does not resolves that second example.

Instead, in addition, we propose to limit subsumption to cases involving an equivalent pack. That way, even though we don't know whether the pack expansion would be empty, both constraints will have the same number of expansions such that subsumption and satisfaction would give consistent results.

We do not reintroduced mixed-operators subsumption, because given

```
template <typename... T>
void f() requires (C<T> || ...); // #1
template <typename... T>
void f() requires (C<T> && ... ); // #2

f();
```

In this example, #2 is more constrained than #1, and satisfaction would be true for #1 and not for #2. Which is consistent. However `true` does not imply `false`, and to preserve the idea that subsumption forms an implication relationship, we do not restore mixed-operators subsumption.

Short circuiting

To be consistent with conjunction constraint and disjection constraints, we propose that fold expanded constrait short circuit (both their evaluation and substitution).

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 concept patterns we need to decompose each concept 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

[temp.constr.constr]

[temp.constr.constr.general]

A *constraint* is a sequence of logical operations and operands that specifies requirements on template arguments. The operands of a logical operation are constraints. There are ~~three~~ four different kinds of constraints:

- conjunctions [temp.constr.op],
- disjunctions [temp.constr.op], ~~and~~
- atomic constraints [temp.constr.atomic], and

- fold expanded constraints [temp.constr.fold].

In order for a constrained template to be instantiated [temp.spec], its associated constraints [temp.constr.decl] shall be satisfied as described in the following subclauses. [*Note:* Forming the name of a specialization of a class template, a variable template, or an alias template [temp.names] requires the satisfaction of its constraints. Overload resolution [over.match.viable] requires the satisfaction of constraints on functions and function templates. — *end note*]

[*Editor's note:* Add after [temp.constr.atomic]]

🔲 Fold expanded constraint [temp.constr.fold]

A *fold expanded constraint* is formed from a constraint C, and a *fold-operator* which can either be `&&` or `||`.

A fold expanded constraint is a pack expansion [temp.variadic]. Let N be the number of elements in the pack expansion parameters [temp.variadic].

A fold expanded constraint whose *fold-operator* is `&&` is satisfied if it is a valid pack expansion and if $N = 0$ or if for each i where $0 \leq i < N$ in increasing order, C is satisfied when replacing each pack expansion parameter with the corresponding ith element. No substitution takes place for any i greater than the smallest i for which the constraint is not satisfied.

A fold expanded constraint whose *fold-operator* is `||` is satisfied if it is a valid pack expansion, $N > 0$, and if for i where $0 \leq i < N$ in increasing order, there is a smallest i for which C is satisfied when replacing each pack expansion parameter with the corresponding ith element. No substitution takes place for any i greater than the smallest i for which the constraint is satisfied.

[*Note:* If the pack expansion expands packs of different size, then it is invalid and the fold expanded constraint is not satisfied. — *end note*]

Two fold expanded constraints are *compatible for subsumption* if their respective constraints both contain an equivalent unexpanded pack [temp.over.link].

[...]

🔲 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*]

- For a *fold-operator* [expr.prim.fold] that is either `&&` or `||`, the normal form of an expression `( ... fold-operator E )` is the normal form of `( E fold-operator...)`.
- For a *fold-operator* that is either `&&` or `||`, the normal form of an expression `( E1 fold-operator ... fold-operator E2 )` is the the normal form of
  - `(E1 fold-operator...) fold-operator E2` if `E1` contains an unexpanded pack, or
  - `E1 fold-operator (E2 fold-operator...)` otherwise.
- The normal form of `(E && ...)` is a fold expanded constraint [temp.constr.fold] whose constraint is the normal form of `E` and whose *fold-operator* is `&&`.
- The normal form of `(E || ...)` is a fold expanded constraint whose constraint is the normal form of `E` and whose *fold-operator* is `||`.
- 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 fold expanded constraint A subsumes another fold expanded constraint B if they are compatible for subsumption, have the same *fold-operator*, and the constraint of A subsumes that of B.

[*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*]

🔲 Variadic templates [temp.variadic]

- In a *fold-expression* [expr.prim.fold]; the pattern is the *cast-expression* that contains an unexpanded pack.
- In a fold expanded constraint [temp.constr.fold]; the pattern is the constraint of that fold expanded constraint.

[...]

[*Editor's note:* Insert after [temp.variadic]/p13]

The instantiation of a *fold-expression* [expr.prim.fold] produces:

- `(` `((`$\texttt{E}_1$ *op* $\texttt{E}_2$`)` *op* $\cdots$`)` *op* $\texttt{E}_N$ `)` for a unary left fold,
- `(` $\texttt{E}_1$ *op* `(`$\cdots$ *op* `(`$\texttt{E}_{N-1}$ *op* $\texttt{E}_N$`))` `)` for a unary right fold,
- `(` `(((E` *op* $\texttt{E}_1$`)` *op* $\texttt{E}_2$`)` *op* $\cdots$`)` *op* $\texttt{E}_N$ `)` for a binary left fold, and
- `(` $\texttt{E}_1$ *op* `(`$\cdots$ *op* `(`$\texttt{E}_{N-1}$ *op* `(`$\texttt{E}_N$ *op* `E)))` `)` for a binary right fold.

In each case, *op* is the *fold-operator*. For a binary fold, `E` is generated by instantiating the *cast-expression* that did not contain an unexpanded pack. [*Example:*

```
template<typename ...Args>
bool all(Args ...args) { return (... && args); }

bool b = all(true, true, true, false);
```

Within the instantiation of `all`, the returned expression expands to `((true && true) && true) && false`, which evaluates to `false`. — *end example*] If N is zero for a unary fold, the value of the expression is shown in [temp.fold.empty]; if the operator is not listed in [temp.fold.empty], the instantiation is ill-formed.

Table 1: Value of folding empty sequences

Operator	Value when pack is empty
`&&`	`true`
`||`	`false`
`,`	`void()`

A fold expanded constraint is not instantiated [temp.constr.fold].

[...]

[*Editor's note:* Insert in Annex C after [diff.cpp23.dcl.dcl]]

[temp.constr]: Template constraints [diff.cpp23.temp.constr]

Change: Some atomic constraints become fold expanded constraints.
Rationale: Permit the subsumption of fold expressions
Effect on original feature: Valid C++23 code may become ill-formed. For example:

```
template <typename ...V> struct A;
struct Thingy {
        static constexpr int compare(const Thingy&) {return 1;}
};

template <typename ...T, typename ...U>
void f(A<T ...> *, A<U ...> *)
requires (T::compare(U{}) && ...);  // was well-formed (atomic constraint of type bool),
                                    // now ill-formed (results in an atomic constraint of type int)
void g(A<Thingy, Thingy> *ap) {
        f(ap, ap);
}
```

Acknowledgments

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

Thanks to Hubert Tong, Jens Mauer and Barry Revzin for their input on the wording.

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