Document number:	*P0485R0*
Date:	*2016-10-16*
Audience:	CWG
Authors:	*Bruno Manganelli* <*bruno.manga95@gmail.com*> *Michael Wong* <*michael@codeplay.com*> *Simon Brand* <*simonrbrand@gmail.com*>
Reply-to	*Bruno Manganelli* <*bruno.manga95@gmail.com*>

Amended rules for Partial Ordering of function templates

This paper aims to modify the definition of a Transformed template (14.5.6.2-3) to better support function parameter packs during partial ordering. It has already reported as CWG 1825. It can support overload resolution for template argument deduction for non-terminal function parameter packs in P0478.

Motivation and Scope

Currently, function parameter packs do not contribute to determining which of two function templates is more specialized than the other.

[*Example 1:*

```
template <class ...T> int f(T*...)  { return 1; }
template <class T>  int f(const T&) { return 2; }
void g() {
   f((int*)0); // Ambiguous, fails in both directions
}
```

-end example]

With the current rules,

14.8.2.4 [temp.deduct.partial] paragraph 8 says,

`If A was transformed from a function parameter pack and P is not a parameter pack, type deduction fails.`

the above call is ambiguous because deduction fails in both directions, with A being T and P being T* in one direction and A being T* and P being T.

This is an unfortunate limitation which has already been pointed out by core issue 1825 (from which the previous example is taken), whose resolution is the base of this proposal.

Proposal

We propose to add an additional step when generating a transformed template for partial ordering involving parameter pack transformations.

When performing partial ordering between two function templates, let FTA be the function template undergoing transformation and FTP be the function template against which template argument deduction is being performed.

If one of the parameter declarations of FTA is a parameter pack in a deduced context, we want to repeat it N consecutive times, where N is the required size such that the number of parameter declarations of FTA and FTP is the same.
If FTP is also template variadic, the minimum value of N is required to be 1, otherwise N is allowed to be 0.

The invented template parameters are appended to the end of the template parameter list.

For example, in the following code,

```
template <class A, class B, class C>
    void foo(A, B, C); // A

template <class A, class... B>
    void foo(A, B*...); // B
```

when creating the transformed template for B against A, the pack substitution would ensure the code behaves as if the declaration of foo (B) had the form

```
template <class A, class... B, class _Inv0>
    void foo(A, _Inv0*, _Inv0*);
```

If type deduction succeeds in both directions and neither candidate is more specialized than the other, the following steps are added:

- A non-variadic function template is considered more specialized than a variadic template
- A function template with a greater number of parameter declarations is considered more specialized than a function template with fewer parameters.

This will ensure that the following example from the standard will continue to be valid.
[Example -

```
template<class... Args> void f(Args... args);                  // #1
template<class T1, class... Args> void f(T1 a1, Args... args); // #2
template<class T1, class T2> void f(T1 a1, T2 a2);             // #3

f();        // calls #1
f(1, 2, 3); // calls #2
f(1, 2);    // calls #3; non-variadic template #3 is more
            // specialized than the variadic templates #1 and #2
```

— end example]

Implementation experience

The changes have been implemented in clang, as part of a proposal to extend template argument deduction to non-terminal function parameter packs (P0478).
https://github.com/bmanga/clang

Standarese Wording available on request.

Reference

CWG 1825: http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1825
P0478: Template argument deduction for non-terminal function parameter packs

Acknowledgement

We like to thank Gordon Brown for his review and suggestions.