James Widman
ISO/IEC JTC1 SC22 WG21 N3281=11-0051 - 2011-03-25

692. Partial ordering of variadic class template partial specializations

Drafting note: The previous version of this proposal made it so that partial ordering acted on all pairs of function parameters (and not only those for which there is an explicit call argument); Jason Merrill argued persuasively against this (by the principle that the effect of partial ordering for a call should mimic the apparent effect of overload resolution between non-template functions where some parameters do not have explicit call arguments).

So for a call context, the new wording below considers only those parameters that have a corresponding explicit call argument.

Proposed resolution:

- [drafting note: The explicit mention of the specific types used for deduction in p4 ("...param list, or in the case of a conv. function...") is unnecessary (because this is spelled out in 14.8.2.4 temp.deduct.partial) and incomplete (because it doesn't mention the specialization declaration case)]

  Change 14.5.6.2 temp.func.order as indicated:

  4. Using the transformed function template's function ~~parameter list, or in the case of a conversion function its transformed return~~ type, perform type deduction against the ~~function parameter list (or return type) of the other function. The mechanism for performing these deductions is given in~~ **other template as described in** 14.8.2.4 temp.deduct.partial. *[Example:* [...] *— end example]*
  5. ***[Note:*** ~~The presence of unused ellipsis and default arguments has no effect on the partial ordering of function templates.~~ **Since partial ordering in a call context considers only parameters for which there are explicit call arguments, some parameters are ignored (namely, function parameter packs, parameters with default arguments, and ellipsis parameters).** *[Example:*

```
template<class T> void f(T);              // #1
template<class T> void f(T*, int=1);      // #2
template<class T> void g(T);              // #3
template<class T> void g(T*, ...);        // #4

  int main() {
    int* ip;
    f(ip);              // calls #2
    g(ip);              // calls #4
  }
```

— end example] ***[Example:***

```
template<class T, class U> struct A { };

template<class T, class U> void f( U, A<U,T>* p = 0 ); // #1
template<         class U> void f( U, A<U,U>* p = 0 ); // #2
template<class T>             void g( T, T = T() );       // #3
template<class T, class ...U> void g( T, U... );          // #4

void h() {
  f<int>( 42, (A<int,int>*)0 );  // calls #2
  f<int>( 42 );                  // Error: ambiguous
  g(42);                         // Error: ambiguous
}
```

— end example] ***[Example:***

```
template <class T, class... U> void f (T, U...); // #1
template <class T>             void f (T);       // #2
template <class T, class... U> void g (T*, U...); // #3
template <class T>             void g (T);        // #4
void h(int i) {
    f(&i); // Error: ambiguous
    g(&i); // Ok, calls #3
}
```

— end example] — end note]

- Change 14.8.2.4 temp.deduct.partial p3 and p4 as indicated:

3. The types used to determine the ordering depend on the context in which the partial ordering is done:
   - In the context of a function call, the **types used are those** function parameter types **for which the function call has arguments** ~~are used~~. ***[Footnote*. Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected. — *end footnote .]***
   - In the context of a call to a conversion operator, the return types of the conversion function templates are used.
   - In other contexts (14.5.6.2) the function template's function type is used.
4. Each type **nominated above** from the parameter template and the corresponding type from the argument template are used as the types of P and A.

- Change 14.8.2.4 temp.deduct.partial p8 (now p9) as indicated:

8. **If `A` was transformed from a function parameter pack and `P` is not a parameter pack, type deduction fails. Otherwise,** using the resulting types `P` and `A` the deduction is then done as described in 14.8.2.5 temp.deduct.type. **If `P` is a function parameter pack, the type `A` of each remaining parameter type of the argument template is compared with the type `P` of the *declarator-id* of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack.** If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template. ***[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]

[Drafing note: some of the new normative text for p8 above was copied from similar wording about parameter packs from 14.8.2.1 temp.deduct.call p1]

- Change 14.8.2.5 temp.deduct.type p9 and p10 as indicated:

9. If `P` has a form that contains `<T>` or `<i>`, then each argument P_i of the respective template argument list of `P` is compared with the corresponding argument A_i of the corresponding template argument list of `A`. If the template argument list of `P` contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If P_i is a pack expansion, then the pattern of P_i is compared with each remaining argument in the template argument list of `A`. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by P_i. **During partial ordering (14.8.2.4 [temp.deduct.partial]), if A_i was originally a pack expansion:**
   - **if `P` does not contain a template argument corresponding to A_i then A_i is ignored;**
   - **otherwise, if P_i is not a pack expansion, template argument deduction fails.**

[Example:

```
template <class T1, class ...Z> class S;                          // #1
template <class T1, class ...Z> class S<T1, const Z&...> {};      // #2
template <class T1, class T2> class S<T1, const T2&> {};          // #3
S<int, const int&> s;  // both #2 and #3 match; #3 is more specialized

template<class T, class... U>             struct A  {};              // #1
template<class T1, class T2, class... U> struct A<T1,T2*,U...> {}; // #2
template<class T1, class T2>              struct A<T1,T2>  {};      // #3

template struct A<int,int*>; // Selects #2
```

— end example]

10. Similarly, if `P` has a form that contains `(T)`, then each parameter type P_i of the respective *parameter-type-list* of `P` is compared with the corresponding parameter type A_i of the corresponding *parameter-type-list* of `A`. If `P` and `A` are function types that originated from deduction when taking the address of a function template (14.8.2.2 temp.deduct.funcaddr) or when deducing template arguments from a function declaration (14.8.2.6 temp.deduct.decl) and P_i and A_i are parameters of the top-level *parameter-type-list* of `P` and `A`, respectively, P_i is adjusted if it is an rvalue reference to a cv-unqualified template parameter and A_i is an lvalue reference, in which case the type of P_i is changed to be the template parameter type (i.e., `T&&` is changed to simply `T`). *[Note*. As a result, when P_i is `T&&` and A_i is `X&`, the adjusted P_i will be `T`, causing `T` to be deduced as `X&` — *end note* . *[Example:*

```
template <class T> void f(T&&);
template <> void f(int&) { }  // #1
template <> void f(int&&) { } // #2
void g(int i) {
  f(i);                       // calls f<int&>(int&), i.e., #1
  f(0);                       // calls f<int>(int&&), i.e., #2
}
```

— end example]

If the *parameter-declaration* corresponding to P_i is a function parameter pack, then the type of its *declarator-id* is compared with each remaining parameter type in the *parameter-type-list* of `A`. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. **During partial ordering (14.8.2.4 [temp.deduct.partial]), if A_i was originally a function parameter pack:**

- **if `P` does not contain a function parameter type corresponding to A_i then A_i is ignored;**
- **otherwise, if P_i is not a function parameter pack, template argument deduction fails.**

[Example:

```
template<class T, class... U> void f(T*, U...){} // #1
template<class T>             void f(T){}        // #2

template void f(int*); // Selects #1
```

— end example] ~~[Note: A function parameter pack can only occur at the end of a *parameter-type-list* (8.3.5 [dcl.fct]). — end note]~~

- Remove 14.8.2.5 temp.deduct.type p22 (because its contents are better expressed in the new wording for 14.8.2.4 temp.deduct.partial):

22. ~~If the original function parameter associated with `A` is a function parameter pack and the function parameter associated with `P` is not a function parameter pack, then template argument deduction fails. *[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]