Talking to Typelists

Type sequences implemented with variadic templates have largely supplanted the traditional use of typelists in new C++ code. However, we have nearly two decade's worth of complex, debugged, and useful legacy typelist code that we'd like to leverage for use with type sequences without having to rewrite them.

Additionally, modern C++ metaprogramming makes extensive use of index sequences as well as type sequences. Many index sequence meta-algorithms are logically similar to corresponding meta-algorithms on typelists (and type sequences). Wouldn't it be convenient to simply and automatically convert a typelist meta-algorithm into a functionally-similar index sequence meta-algorithm?

To be concrete, if we have `Sort` meta-algorithm that sorts a typelist with a compile-time comparator that compares types, wouldn't it be nice to generate automatically a `sort_is` meta-algorithm that sorts an index sequence with a compile-time comparator that compares indexes? You bet it would!

```
template <typename IntegerSequence,
          template <size_t, size_t> class Comparator>
using sort_is = convert<IntegerSequence, Comparator, Sort>;
```

That's what we're going to do in this installment of Once, Weakly.

First, we'll show how to translate among typelists, type sequences, and index sequences so that each of these sequence categories can be manipulated with meta-algorithms written for other sequence categories.

Then, we'll show how to adapt predicates and comparators for indexes to be converted to similar predicates and comparators on types.

Finally, we'll show how to automate the adaptation of a typelist meta-algorithm to an index sequence meta-algorithm while simultaneously adapting predicates and comparators.

Typelists

A traditional typelist is a nested type structure that represents an ordered sequence of zero or more types. They look like this:

```
template <class H, class T> // a type followed by a type list, or...
struct typelist {
    typedef H head;
    typedef T tail;
};
class null_typelist {};     // ...an empty type list
```

A typelist meta-algorithm performs some useful compile-time calculation on a typelist. Here's an STL-like partition algorithm for a typelist.[1]

```
template <class TList, template <class> class Pred>
struct Partition;
```

The `Pred` parameter is a predicate over a type. For instance:

```
template <class T> struct IsSmall
    { enum { value = sizeof(T) < 8 }; };
```

Assuming we have available an initial typelist, we can partition it according to whether its members are "small":[2]

```
typedef typename Partition<initial, IsSmall>::type partitioned;
```

The implementation of `Partition` is relegated to two partial specializations of the primary template. The first handles the degenerate case of a null typelist:[3]

```
template <template <class> class Pred>
struct Partition<null_typelist, Pred> {
    typedef null_typelist type;
    enum { value = 0 };
};
```

The second performs recursive specialization to reorder the typelist according to the predicate:

```
template <class Head, class Tail, template <class> class Pred>
struct Partition<typelist<Head, Tail>, Pred> {
    typedef typename Select<
        Pred<Head>::value,
        typelist<Head, typename Partition<Tail, Pred>::type>,
        typename Append<typename Partition<Tail, Pred>::type,
                        Head>::type
    >::type type;
    enum { value
        = Partition<Tail, Pred>::value + Pred<Head>::value };
};
```

[1] For more about typelists, see Andrei Alexandrescu's *Modern C++ Design*, Addison-Wesley 2001. For examples of C++03 typelist meta-algorithms and meta-algorithm implementations, see Once, Weakly for 9 September 2003 and 21 February 2004.

[2] https://en.wikipedia.org/wiki/Let%27s_Get_Small.

[3] Here, we use the modern convention of giving a type result the name "type" and a value result the name "value." The code accompanying the Once, Weakly installments mentioned above use an older, idiosyncratic convention of "R" for a type result and "r" for a value result. The code for the typelist meta-algorithms that accompany this installment of Once, Weakly uses the modern convention.

Type Sequences

In modern C++ we prefer to use type sequences defined by a variadic template with a typename parameter pack:

```
template <typename... Ts>
struct type_sequence {};
```

As with typelists, we have frequent recourse to meta-algorithms on type sequences. For example, we may also want to partition a type sequence:[4]

```
using initial = type_sequence<~~~>;
using partitioned = partition_t<initial, IsSmall>;
```

`partition_t` is a convenience using-declaration:

```
template <typename TypeSequence, template <typename> class Pred>
struct partition;

template <typename TypeSequence, template <typename> class Pred>
using partition_t = typename partition<TypeSequence, Pred>::type;
```

The implementation of partition for type sequences is a simple upgrade of the typelist version. An empty type sequence is easy to partition:

```
template <template <typename> class Pred>
struct partition<type_sequence<>, Pred> {
    using type = type_sequence<>;
    enum : size_t { value = 0 };
};
```

The main body of the implementation uses the omnipresent First/Rest idiom:

```
template <typename H, typename... T, template <typename> class Pred>
struct partition<type_sequence<H, T...>, Pred> {
    using Ptail = partition<type_sequence<T...>, Pred>;
    using tail_t = typename Ptail::type;

    using type = std::conditional_t<
        Pred<H>::value,
        pushfront_t<H, tail_t>,
        pushback_t<H, tail_t>
    >;
    enum : size_t { value = Ptail::value + Pred<H>::value };
};
```

[4] Note the convention in the code examples of using three consecutive tildes to indicate that details have been elided. The more conventional use of an ellipsis is confusing in modern C++, where we have ellipses coming out of our ears, and the character sequence ~~~ is not (yet!) syntactically-correct C++.

So, it was straightforward but non-trivial to re-implement the typelist partition meta-algorithm as a type sequence meta-algorithm. However, other meta-algorithms are not so trivial, and (of course) it's always inadvisable to repeat oneself by establishing parallel bodies of code which must be maintained in sync.

It would be preferable to leverage our existing typelist meta-algorithms directly. All we need is a way to translate between typelists and type sequences.

Converting Between Typelists and Type Sequences

Making typelists in modern C++ is trivial...

```
template <typename... Ts>
struct MakeTypelist;

template <typename... Ts>
using make_typelist = typename MakeTypelist<Ts...>::type;

template <typename Head, typename... Tail>
struct MakeTypelist<Head, Tail...> {
    using type = typelist<Head, make_typelist<Tail...>>;
};

template <>
struct MakeTypelist<> {
    using type = null_typelist;
};
```

...and thanks to parameter pack expansion, converting a type sequence to a typelist is even easier.

```
template <typename TypeSequence>
struct TS_2_TL;

template <typename TypeSequence>
using type_sequence_2_typelist
    = typename TS_2_TL<TypeSequence>::type;

template <typename... Ts>
struct TS_2_TL<type_sequence<Ts...>> {
    using type = make_typelist<Ts...>;
};
```

Of course, we'll want to be able to convert a typelist back to a type sequence:[5]

```
template <typename Typelist>
struct TL_2_TS;
```

[5] The implementation is unsurprising but lengthy and may be found in the accompanying code.

```
template <typename Typelist>
using typelist_2_type_sequence = typename TL_2_TS<Typelist>::type;
```

Leveraging Legacy Type List Meta-Algorithms

As an example of a legacy typelist meta-algorithm we'd like to leverage for type sequences, consider a set intersection that finds the intersection of two unordered typelists:

```
template <class TList1, class TList2>
struct SetIntersection;
```

In order to find the set intersection of two type sequences, all we have to do is convert the type sequences to typelists, intersect the typelists, and convert the resulting typelist to a type sequence.

First, get/make/find/steal some type sequences:

```
using s1 = ~~~;
using s2 = ~~~;
```

Convert them to typelists:

```
using t1 = type_sequence_2_typelist<s1>;
using t2 = type_sequence_2_typelist<s2>;
```

Get their intersection as a typelist:

```
using inter = typename SetIntersection<t1, t2>::type;
```

Convert back to a type sequence:

```
using intersection = typelist_2_type_sequence<inter>;
```

That's quite a mouthful! A using-declaration can improve readability.

```
template <typename TSeq1, typename TSeq2>
using set_intersection_ts =
    typelist_2_type_sequence<
        typename SetIntersection<
            type_sequence_2_typelist<TSeq1>,
            type_sequence_2_typelist<TSeq2>
        >::type
    >;
```

That's quite a mouthful too but—as usual—it will benefit the users of our code.

```
using intersection = set_intersection_ts<s1, s2>;
```

So, we have the useful but rather unsurprising result that we can leverage meta-algorithms written for lists of types to work with sequences of types. Let's turn now to the possibility of manipulating other kinds of sequences with typelist meta-algorithms.

Index Sequences

C++14 has a standard index sequence template and a means to generate index sequences. It looks something like this:

```
template <size_t... indexes>   // it's not exactly this,
struct index_sequence { ~~~ }; // but close enough

using seq = make_index_sequence<N>; // [0, N)
```

We'll discuss in more detail why index sequences are so useful in combination with pack expansion in a future Once, Weakly. For now, let's just note that it's useful to be able to generate a variety of different index sequences through the application of index sequence meta-algorithms.

But do we really have to write index sequence meta-algorithms? After all, nearly anything we might want to do with a sequence of indexes has probably already been implemented as a meta-algorithm that works with a type sequence or, more probably, a typelist. All we need to provide is a means to translate between index sequences and type sequences.

Translating Between Index Sequences and Type Sequences

First, let's find a way to (take your pick) wrap a type around an index, or convert an index into a type. This is a long-solved problem:[6]

```
template <size_t i>
struct Index2Type {
    enum : size_t { value = i };
};
```

The interface is similar to what we've already seen.

```
template <typename IndexSequence>
struct IS_2_TS;

template <typename IndexSequence>
using index_sequence_2_type_sequence
    = typename IS_2_TS<IndexSequence>::type;
```

Pack expansion renders the implementation trivial:

```
template <size_t... iseq>
struct IS_2_TS<index_sequence<iseq...>> {
    using type = type_sequence<Index2Type<iseq>...>;
};
```

The implementation of a capability to revert such a type sequence to an index sequence is equally straightforward.

[6] Andrei again. I mean, *op cit*. There, it's called `Int2Type`.

It's probably clear at this point that we can combine these conversion operations to allow most conversions among these three sequence types:

```
index_sequence_2_type_sequence
type_sequence_2_typelist
typelist_2_type_sequence
revert_type_sequence_2_index_sequence
index_sequence_2_typelist
typelist_2_index_sequence
```

This in turn means that we can now find the set intersection of two index sequences, using a meta-algorithm written for typelists (or type sequences, for that matter).

Get/make/find/steal some index sequences:

```
using s1 = make_index_sequence<N>;
using s2 = make_index_interval<5, N+5>;
```

Convert them to typelists:

```
using t1 = index_sequence_2_typelist<s1>;
using t2 = index_sequence_2_typelist<s2>;
```

Intersect the typelists:

```
using inter = typename SetIntersection<t1, t2>::type;
```

And convert the resulting typelist to an index sequence:

```
using inter_indexes = typelist_2_index_sequence<inter>;
```

Sequence Predicates and Comparators

The typelist set intersection meta-algorithm works for converted index sequences because there is a 1-1 mapping between each index and the type generated by `Index2Type`. Because this set intersection algorithm compares elements for equality (rather than equivalence) the 1-1 mapping ensures that if two generated types compare equal, the underlying indexes must as well.

Not all typelist meta-algorithms have this property, and are instead parameterized with type predicates. However, if we're actually processing an index sequence, the predicate or question we want to ask will be about indexes, not types.

We can address this issue by making an index predicate look like a type predicate.

```
template <template <size_t> class IndexPred>
struct IndexPred2TypePred {
    template <typename T>
    struct TypePred {
        enum : bool { value = IndexPred<T::value>::value };
    };
};
```

The index predicate is disguised as a type predicate, but reverts to an index predicate when invoked with a type generated from an index with `Index2Type`.

As an example, consider a simple find-if meta-algorithm on typelists:

```
template <class TL, template <class> class Pred>
struct FindIf;
```

The meta-algorithm expects to be invoked with a typelist and a type predicate. We have to convert the index sequence to a typelist, and an index predicate to a type predicate. For example…

```
template <size_t i>
struct IsOdd : truthiness<i&1> {};
```

…is an index predicate that tells us whether an index is odd or not. The base class `truthiness`[7] could be defined as

```
template <bool c>
using truthiness
    = std::conditional_t<c, std::true_type, std::false_type>;
```

Now we can look for odd indexes. First, find or steal an index sequence:

```
using iseq = index_sequence<0,2,6,2,4,9,6,7>;
```

Convert it to a typelist:

```
using tl = index_sequence_2_typelist<iseq>;
```

Convert index predicate to type predicate:

```
template <typename T>
using IsOdd_t = IndexPred2TypePred<IsOdd>::TypePred<T>;
```

Invoke the typelist meta-algorithm:

```
constexpr size_t index = FindIf<tl, IsOdd_t>::value; // == 5
```

Similarly, we may want to sort an index sequence using a typelist sort meta-algorithm.

```
template <class TL, template <class, class> class Comp>
class Sort;
```

As with our index predicate, we have to be able to disguise an index comparator as a type comparator:

```
template <template <size_t, size_t> class IndexComp>
struct IndexComp2TypeComp {
    template <typename T1, typename T2>
    struct TypeComp {
```

[7] http://cs.union.edu/seminar/archive/2008-9/saks.html.

```
        enum : bool
            { value = IndexComp<T1::value, T2::value>::value };
    };
};
```

Similarly,

```
template <typename T1, typename T2>
using IsGreater_t = IndexComp2TypeComp<IsGreater>::TypeComp<T1, T2>;
~~~
using is = index_sequence<3, 2, 1, 8, 4, 7, 5, 6, 9>;
using tl = index_sequence_2_typelist<is>;
using result_tl = typename Sort<tl, IsGreater_t>::type;
using result = typelist_2_index_sequence<result_tl>;
// == index_sequence<9, 8, 7, 6, 5, 4, 3, 2, 1>;
```

Useful Use of Using

That's yet another mouthful. As usual, we can call on one or more using-declarations to simplify the syntax:

```
template <typename Typelist,      // sort a typlist
          template <typename, typename> class Pred>
using sort_tl = ~~~;

template <typename TypeSequence,  // sort a type sequence
          template <typename, typename> class Pred>
using sort_ts = ~~~;

template <typename IndexSequence, // sort an index sequence
          template <size_t, size_t> class Pred>
using sort_is = ~~~;
```

Now it's easy to sort typelists, type sequences, and index sequences with the same legacy typelist meta-algorithm:

```
using result_tl = sort_tl<tl, IsGreater_t>; // sort typelist
using result_ts = sort_ts<ts, IsGreater_t>; // sort type sequence
using result_is = sort_is<is, IsGreater>;   // sort index sequence
```

Unfortunately, the implementation of (for example) `sort_is` is not the kind of code we'd like to have to repeat each time we convert a typelist meta-algorithm:

```
template <
   typename IndexSequence,
   template <size_t, size_t> class Comp>
using sort_is =
    typelist_2_index_sequence<
        typename Sort<
```

```
            index_sequence_2_typelist<IndexSequence>,
            IndexComp2TypeComp<Comp>::template TypeComp
        >::type
    >;
```

Fortunately, we can promote the hard-coded typelist meta-algorithm (`Sort`, in this case) to a parameter of the using-declaration:

```
template <
    typename IndexSequence,
    template <size_t, size_t> class Comp,
    template <typename,
              template <typename, typename> class> class Algorithm>
using convert_algorithm_1_comp =
    typelist_2_index_sequence<
        typename Algorithm<
            index_sequence_2_typelist<IndexSequence>,
            IndexComp2TypeComp<Comp>::template TypeComp
        >::type
    >;
```

We can produce a more-or-less complete set of these Adapter[8] pattern-inspired `using`s to automate conversion of meta-algorithms/predicates/comparators for one sequence type to work with another sequence type. For example, to convert typelist meta-algorithms for use with index sequences, we might have:

```
template < // typelist meta-algorithm takes a single typelist.
    typename IndexSequence,
    template <typename> class Algorithm>
using convert_algorithm_1 = ~~~;

template < // here it takes two typelists.
    typename IndexSequence1,
    typename IndexSequence2,
      template <typename, typename> class Algorithm>
using convert_algorithm_2 = ~~~;

template < // here it takes a single typelist and a predicate.
    typename IndexSequence,
    template <size_t> class Pred,
    template <typename, template <typename> class> class Algorithm>
using convert_algorithm_1_pred = ~~~;

template < // here it takes a single typelist and a comparator.
    typename IndexSequence,
```

[8] http://c2.com/cgi/wiki?AdapterPattern.

```
    template <size_t, size_t> class Comp,
    template <typename,
              template <typename, typename> class> class Algorithm>
using convert_algorithm_1_comp = ~~~;

template < // here it takes two typelists and a predicate.
    typename IndexSequence1,
    typename IndexSequence2,
    template <size_t> class Pred,
    template <typename, typename,
              template <typename> class> class Algorithm>
using convert_algorithm_2_pred = ~~~;

template < // here it takes two typelists and a comparator.
    typename IndexSequence1,
    typename IndexSequence2,
    template <size_t, size_t> class Comp,
    template <typename, typename,
              template<typename, typename> class> class Algorithm>
using convert_algorithm_2_comp = ~~~;
```

With these adapters in place it's easy to leverage our existing set of typelist meta-algorithms for manipulating index sequences:

```
template <typename ISeq,
          template <size_t, size_t> class Comp>
using sort_is =
    convert_algorithm_1_comp<ISeq, Comp, Sort>;

template <typename ISeq,
          template <size_t> class Pred>
using partition_is =
    convert_algorithm_1_pred<ISeq, Pred, Partition>;

template <typename ISeq1, typename ISeq2>
using set_intersection_is =
    convert_algorithm_2<ISeq1, ISeq2, SetIntersection>;
```

Going the Other Way

It's easy to leverage meta-algorithms on typelists or type sequences to work with index sequences because, with a facility like `Index2Type`, it's straightforward to construct a mapping from indexes to types and back again.

What about going the other way? That is, can we use index sequence meta-algorithms on typelists or type sequences? It's possible, but the implementation of the mapping from

type to integral is much more complex, and typically imperfect (at least without assistance from the compiler, which is unlikely to be forthcoming).

For an example of an approach to such a mapping, see a Gödel mapping of the C++ type system in CUJ.[9]

Next Time…

In addition to sequences of types and sequences of indexes, it's profitable to manipulate sequences of other kinds of entities. In the next installment of Once, Weakly we'll look at template sequences and show how we can leverage meta-algorithms on typelists and type sequences to manipulate them as well.

stevedewhurst.com

[9] Stephen C. Dewhurst, A Bit-Wise Typeof Operator, Part 1. C/C++ Users Journal 20, 8 (August 2002), A Bit-Wise Typeof Operator, Part 2. C/C++ Users Journal 20, 10 (October 2002), and A Bit-Wise Typeof Operator, Part 3. C/C++ Users Journal 20, 12 (December 2002).