Document number:	P0649R0
Date:	2017-06-15
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Reflection Working Group / Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

Other Product-Type algorithms

Abstract

This paper proposes some algorithms based on *ProductType* P0327R2 and *TypeConstructible* P0338R1 proposals.

Table of Contents

History

R0

Take in account the feedback from Kona meeting concerning P0327R1. Next follows the direction of the committee: Split the proposal into 3 documents

- Product Type Access
- Adaptation of current tuple-like algorithms to *ProductType*
- Other *ProductType* algorithms

In this document, we describe some additional basic algorithms applicable to *ProductTypes*. There are yet a lot of them.

Introduction

There are a lot of algorithms working on *ProductType* P0327R2; a lot of the homogeneous container algorithm are applicable to heterogeneous containers and functions, see Boost.Fusion and Boost.Hana.

P0648R0 proposes the basic algorithm that could be used in the definition of the extension of some tuple-like algorithm already defined on the standard as `apply`, `swap`, `lexicographical_compare`, `cat`, `assign`, `move`, ...

Some examples of such algorithms are `for_each`, `find`, `fold`, `any_of`, `all_of`, `none_of`, `accumulate`, `count`, ...

Other algorithms that need in addition that the *ProductType* to be also *TypeConstructible* are e.g. `transform`, `filter`, `replace`, `zip`, `flatten`, ...

Motivation

Other functions for *ProductType*

Aside P0648R0 there are a lot of useful function associated to product types that make use only of the product type access traits and functions.

`for_each`

```
template <class F, class ProductType>
  constexpr void for_each(F&& f, ProductType&& pt);
```

This is the equivalent of `std::for_each` applicable to product types instead of homogeneous containers or range types.

In the absence of product-type based for loops, this function will cover the hole.

`fold_left` / `fold_right` / `accumulate`

This is the equivalent of `std::accumulate` applicable to product types instead of homogeneous containers types.

It has the same motivation.

`all_of`

Checks if 1-unary p-predicate `p` returns `true` for all elements in the product type.

A p-predicate is a polymorphic predicate, that is an overload set.

It has the same motivation as the standard functions for homogeneous containers or range types.

`any_of`

Checks if 1-unary p-predicate `p` returns `true` for at least one elements in the product type.

It has the same motivation as the standard functions for homogeneous containers or range types.

`none_of`

Checks if 1-unary predicate `p` returns `true` for no elements in the product type.

It has the same motivation as the standard functions for homogeneous containers or range types.

`hash_value`

This would depend on the new `hash_value/hash_combine` interface as proposed in P0029R0.

Other functions for *TypeConstructible ProductTypes*

`transform`

```
template <class F, class ProductType>
  constexpr `see below` transform(F&& f, ProductType&& pt);
```

This is the equivalent of `std::transform` applicable to product types instead of homogeneous containers types.

This needs in addition that `ProductType` is *TypeConstructible* (See [P0338R0]). Note that `std::pair`, `std::tuple` and `std::array` are *TypeConstructible*, but `std::pair` and `std::array` limit either in the number or in the kind of types (all the ame).

A c-array is not type *TypeConstructible* as it cannot be returned by value.

Proposal

This paper proposes some algorithms that can be built on top of the *ProductType* and the *TypeConstructible* requirements.

Design Rationale

Locating the interface on a specific namespace

The name of *product type* algorithms, `transform`, `replace`, `join`, are quite common. Nesting them on a specific namespace makes the intent explicit.

We can also preface them with `product_type_`, but the role of namespaces was to be able to avoid this kind of prefixes.

If the user want to use shorter name it has always the possibility to define an namespace alias.

```
namespace stdex = std::experimental;
```

or import those into his own namespace

```
namespace mns {
    using namespace std::experimental;
}
```

P-Callable and P-Predicates

The callable and predicate types passed to some algorithms must be polymorphic, as we have heterogeneous types to what it should be applied. The user can use the proposed `overload` function [OVERLOAD] to construct this overload set or use generic lambdas.

N-Callable and N-Predicates

An alternative could be to pass a *ProductType* with a specific *Callable*/*Predicate* to apply on the element type of the *ProductType*. I call those *N-Callable*/*N-Predicate*.

This paper is not proposing the use of *N-Callable*/*N-Predicate*, but the authors are looking for use cases where this could be useful.

This is in relation with Haskell BiFunctor.

Other functions for *TypeConstructible ProductTypes*

Some algorithms need a *TypeConstructible ProductTypes* as they need to construct a new instance of a *ProductTypes*.

An alternative is to use `std::tuple` as the parameter determining the *Product Type* to construct.

We could also add a *TypeConstructible* parameter, as e.g.

```
    template <template <class...> TC, class ...ProductTypes>
        constexpr `see below` cat(ProductTypes&& ...pts);
    template <class TC, class ...ProductTypes>
        constexpr `see below` cat(ProductTypes&& ...pts);
```

Where `TC` is a variadic template for a *ProductType* as e.g. `std::tuple` or a TypeConstructor P0343R0.

Shouldn't some of these functions belong to another more generic type of classes?

Most of the proposed algorithms for *ProductType* correspond to a more generic type of classes. E.g. `transform`, is associated to *Functor*. The proposed algorithms correspond to the customization.

However some algorithms are not part of the customization point of the more generic type of classes, and defining them here is a loss of time if we couldn't be able to customize them.

Waiting for those more general type of classes, we propose to add them here as we consider than the implementation for a *ProductType* could have a better complexity and perform better.

Proposed Wording

The proposed changes are expressed as edits to N4564 .

Add the following section in N4564

Product type algorithms

Some algorithms need a `make<TC>(args...)` factory P0338R1.

If the first product type argument is *TypeConstructible* from the resulting `Types` then return an instance of it; otherwise construct a `std::tuple`.

Product type algorithms synopsis

```
namespace std {

namespace product_type {

    template <class ProductType>
        constexpr bool is_empty(ProductType&& pt);
    template <class ProductType>
        constexpr auto back(ProductType&& pt);
    template <class ProductType>
        constexpr auto front(ProductType&& pt);

    template <size_t N, class ProductType>
        constexpr auto drop_front(ProductType&& pt);
    template <size_t N, class ProductType>
        constexpr auto drop_back(ProductType&& pt);

    template <size_t I, class ProductType, class T>
        constexpr auto insert(ProductType&& pt, T&& x);

    template <class F, class State, class ProductType
        constexpr State fold_left(ProductType&& pt, State&& state, F&& f);
    template <class F, class ProductType
        constexpr auto fold_left(ProductType&& pt, F&& f);

    template <class F, class ProductType
        constexpr void for_each(ProductType&& pt, F&& f);

    template <class ProductType, class F>
        constexpr bool transform(ProductType&& pt, F&& f);


}}
```

Function Template `product_type::fold_left`

```
template <class F, class State, class ProductType>
  constexpr State fold_left(ProductType&& pt, State&& state, F&& f);

template <class F, class ProductType
  constexpr State fold_left(ProductType&& pt, F&& f);
```

Function Template `product_type::is_empty`

```
template <class ProductType>
  constexpr bool is_empty(ProductType&& pt);
```

Returns `product_type::size<ProductType> == 0` .

Function Template `product_type::front`

```
template <class ProductType>
  constexpr auto front(ProductType&& pt);
```

Requires the `ProductType` `pt` is not empty.

Returns The first element of `pt` .

Function Template `product_type::back`

```
template <class ProductType>
  constexpr auto back(ProductType&& pt);
```

Requires the `ProductType` `pt` is not empty.

Returns The last element of `pt` .

Function Template `product_type::transform`

```
template <class ProductType, class F>
  constexpr bool transform(ProductType&& pt, F&& f);
```

Requires: `F` is Callable with each one of the `ProductType` elements.

Returns: A *ProductType* constructed with the same type_constructor than the *ProductType* `ProductType` rebinding each element with the result type of the application of `F` to each element.

Function Template `product_type::drop_front`

```
template <size_t N, class ProductType>
  constexpr auto drop_front(ProductType&& pt);
```

Returns Drop the first `N` elements of `pt` and return the product type of the other elements in the same order.

Remarks This function should not participate in overload resolution if the *ProductType* `ProductType` is not able to rebind with the not dropped elements.

Note `std::tuple` and `std::array` are able to do that, but not `std::pair` or any user defined struct.

Function Template `product_type::drop_back`

```
template <size_t N, class ProductType>
  constexpr auto drop_back(ProductType&& pt);
```

Returns Drop the last `N` elements of `pt` and return the product type of the other elements in the same order.

Remarks This function should not participate in overload resolution if the *ProductType* `ProductType` is not able to rebind with the not dropped elements.

Note `std::tuple` and `std::array` are able to do that, but not `std::pair` or any user defined struct.

Function Template `product_type::insert`

```
template <size_t I, class ProductType, class T>
  constexpr auto insert(ProductType&& pt, T&& x);
```

Returns: Insert a value at a given index in a *ProductType*. Given a *ProductType* `pt`, an index `I` and an element to insert `x`, `insert` inserts the element at the given index.

Remarks This function should not participate in overload resolution if the *ProductType* `ProductType` is not able to rebind with the not resulting elements.

Note `std::tuple` and `std::array` are able to do that, but not `std::pair` or any user defined struct.

Implementability

This is a library proposal. There is an implementation PT_impl of the basic *ProductType* algorithms. Not all the proposed algorithms have been implemented.

Open Points

The authors would like to have an answer to the following points if there is any interest at all in this proposal:

- Do we want this for Fundamental TS V3?

Future work

Add other algorithms on *Product Types*

See Boost.Hana documentation.

Searchable algorithms:

- contains
- in
- find
- find_if
- is_disjoint
- is_subset

Sequence algorithms:

- cartesian_product
- group
- insert_range
- interperse
- partition
- permutations
- remove_at
- remove_range
- reverse
- scan_left
- scan_right
- slice

- sort
- ...

Product Types views and lazy algorithms

Based on Range views for homogeneous Ranges Range-v3, views for heterogeneous sequences Boost.Fusion, Boost.Hana define *Product Types* views, adaptors, ...

Tagged *Product Types*

Based on the work N4569 for tagged tuples, associative sequences in Boost.Fusion, Struct in Boost.Hana define Tagged *ProductTypes* and specific algorithms for them.

Acknowledgments

Thanks to all those that helped on P0327R1.

Thanks to Louis Ideone for his wonderful Boost.Hana library.

Special thanks and recognition goes to Technical Center of Nokia - Lannion for supporting in part the production of this proposal.

References

- Boost.Fusion Boost.Fusion 2.2 library

  http://www.boost.org/doc/libs/1_60_0/libs/fusion/doc/html/index.html

- Boost.Hana Boost.Hana library

  http://boostorg.github.io/hana/index.html

- P0029R0 A Unified Proposal for Composable Hashing

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0029r0.html

- N4564 N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 PDTS

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4564.pdf

- N4569 Proposed Ranges TS working draft

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4569.pdf

- P0327R1 Product Type Access (Revision 1)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0327r1.pdf

- P0327R2 Product Type Access (Revision 2)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0327r2.pdf

- P0338R1 C++ generic factories (Revision 1)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0338r1.pdf

- P0343R0 Meta-programming High-Order Functions

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0343r0.html

- P0648R0 Extending Tuple-like algorithms to Product-Types

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0648r0.pdf

- PT_impl Product types access emulation and algorithms

  https://github.com/viboes/std-make/tree/master/include/experimental/fundamental/v3/product_type

- Range-v3 range-v3

  https://github.com/ericniebler/range-v3