Extending Tuple-like algorithms to Product-Types

Abstract

This paper proposes to adapt the algorithms and interfaces working today with tuple-like types to Product-Types based on P0327R2 proposal.

Table of Contents

- History
- Introduction
- Motivation
- Proposal
- Design Rationale
- Proposed Wording
- Implementability
- Open points
- Future work
- Acknowledgements
- References

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
  - P0327R2 Product Type Access
  - P0648R0 Extension of current tuple-like algorithms to ProductType
  - P0649R0 Other ProductType algorithms

In this document, we describe the extension of the current tuple-like algorithms to the proposed ProductType requirements.

Introduction

There are some algorithms working almost on tuple-like types that can be extended to ProductTypes. Some examples of such algorithms are `apply`, `swap`, `lexicographical_compare`, `cat`, `assign`, `move`, ...

Motivation

Adaptation

Algorithms such as `std::tuple_cat` and `std::apply` that work well with tuple-like types, should work also for any ProductType types.

Reuse

The definition of some the existing functions and assignment operators will surely be implemented using the same algorithm generalized for any ProductType. This paper proposes only the algorithms that could be needed to implement (or define) the current tuple-like interface extended to ProductTypes.

Deprecation
Some of the current algorithms could be deprecated in favor of the new ones. E.g. we could deprecate `std::apply` in favor of `std::product_type::apply`.

Extension

There are many more of them. [P0649R0] takes in account some of the algorithms that work well with ProductTypes.

Proposal

tuple-like algorithms and function adaptation

`std::tuple_cat`

Adapt the definition of `std::tuple_cat` in [tuple.creation] to take care of ProductType.

Note: The single difference between `std::tuple_cat` and `std::product_type::cat` is `std::tuple_cat` forces `std::tuple<>` as result type.

Constructor from a product type with the same number of elements as the tuple

Similar to the constructor from `pair`.

This simplifies a lot the `std::tuple` interface (See [N4387]).

`std::apply`

Adaptation of the definition of `std::apply` to take care of ProductType.

NOTE: This algorithm could just forward to `product_type::apply`.

`std::pair`

piecewise constructor

The following constructor could also be generalized to ProductTypes.

Instead of

```cpp
template <class... Args1, class... Args2>
pair<piecewise_construct_t, 
    tuple<Args1...> first_args, tuple<Args2...> second_args>;
```

we could have

```cpp
template <class PT1, class PT2>
pair<piecewise_construct_t, PT1 first_args, PT2 second_args>;
```

Constructor and assignment from a product type with two elements

Similar to the `tuple` constructor from `pair`.

This simplifies a lot the `std::pair` interface (See [N4387]).

Functions for ProductType

The definition of the existing function will surely be implemented using the same algorithm generalized for any ProductType.

`product_type::apply`

```cpp
template <class F, class ProductType>
constexpr decltype(auto) apply(F&& f, ProductType&& pt);
```

This is the equivalent of `std::apply` applicable to product types instead of tuple-like types.

`std::apply` could be defined in function of it.

`product_type::assign`
Assignment from another product type with the same number of elements and convertible elements.

This function can be used while defining the `operator=` on product types. See the wording changes for `std::tuple`, `std::pair` and `std::array`.

```cpp
template <class PT1, class PT2>
PT1& assign(PT1& pt1, PT2&& pt2);
```

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.

```cpp
template <template <class...> TC, class PT> 
constexpr 'see below' cat(TC PT...); 
```

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

An alternative is to use `std::tuple_cat` when the first `Product Type` is not `Type Constructible` but this creates a cycle.
We could also have an additional parameter stating which will be the result

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

_std::tuple_cat_ could be defined in function of one of the alternatives.

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 same).

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

Design Rationale

Locating the interface on a specific namespace

The name of _product type_ algorithms, _cat_. _apply_. _swap_. _assign_. or _move_ 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.

Proposed Wording

The proposed changes are expressed as edits to [N4564](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0550r0.html) Working Draft, C++ Extensions for Library Fundamentals, Version 2.

If [P0550R0] is accepted, any use of `remove_cv_t<remove_reference_t<T>>` should be replaced by `uncvref_t`.

Add the following section in [N4564](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0550r0.html)

Product type algorithms synopsis

```cpp
namespace std::experimental {
    inline namespace fundamental_v3 {
        namespace product_type {
            template <class F, class ProductType>
                decltype(auto) apply(F&& f, ProductType&& pt);

            template <class PT1, class PT2>
                PT1 assign(PT1&& pt1, PT2&& pt2);

            template <class... PTs>
                `see below` cat(PTs&& ...pts);

            template <class T, class PT>
                `see below` make_from(PT&& pt);

            template <class PT1, class PT2>
                PT1 move(PT1&& pt1, PT2&& pt2);

            template <class PT>
                `see below` to_tuple(PT&& pt);

            template <class PT>
                void swap(PT& x, PT& y) noexcept(`see below`);
        }
    }
}
```

Function Template _product_type::apply_

```cpp
template <class F, class PT>
    constexpr decltype(auto) apply(F&& f, PT&& pt);
```
Effects: Given the exposition-only function:

```cpp
template <class F, class PT, size_t... I>
constexpr decltype(auto) apply_impl(F&& f, PT&& t, index_sequence<I...>) { // exposition only
  return INVOKE(std::forward<F>(f),
                 product_type::get-I<std::forward<Tuple>>(t)...);
}
```

Equivalent to:

```cpp
return apply_impl(std::forward<F>(f), std::forward<PT>(t),
                  product_type::element_sequence_for<PT>[]);
```

Let \( U_i \) be \( \text{product_type}::\text{element}_{t<i}, \text{remove_cv_t}<\text{remove_reference_t}<\text{PT}>> \).

**Function Template** \( \text{product_type}::\text{assign} \)

```cpp
template <class PT1, class PT2>
PT1& assign(PT1& pt1, PT2&& pt2):
```

In the following paragraphs, let \( VPT2 \) be \( \text{remove_cv_t}<\text{remove_reference_t}<\text{PT}>> \), \( Ti \) be \( \text{product_type}::\text{element}_{t<i}, PT1> \) and \( U_i \) be \( \text{product_type}::\text{element}_{t<i}, VPT2> \).

**Requires:** both \( PT1 \) and \( VPT2 \) are ProductTypes with the same size, \( \text{product_type}::\text{size}<PT1>::\text{value}==\text{product_type}::\text{size}<VPT2>::\text{value} \) and \( \text{is_assignable_v}<Ti, \text{const U_i}> \) is true for all \( i \).

**Effects:** Assigns each element of \( pt2 \) to the corresponding element of \( pt1 \).

**Function Template** \( \text{product_type}::\text{cat} \)

```cpp
template <template <class...> TC, class... PTs>
constexpr TC<Types> cat(Types... pts);
```

In the following paragraphs, let \( Ti \) be the \( i \) th type in \( PTs \), \( U_i \) be \( \text{remove_reference_t}<\text{PT}>> \), \( t_i \) be the \( i \) th parameter in the function parameter pack \( pts \), where all indexing is zero-based and

**Requires:** For all \( i \), \( U_i \) shall be the type \( \text{cv_t}<\text{PT_i}>, \text{where}\ \text{cv_t} \) is the (possibly empty) \( i \) th cv-qualifier-seq. Let \( Aik \) be \( \text{product_type}::\text{element}_{t<i}, PT_i> \), the \( ki \) th type in \( PT_i \). For all \( Aik \) the following requirements shall be satisfied: If \( Ti \) is deduced as an lvalue reference type, then \( \text{is_constructible_v}<\text{Aik}, \text{cv_t}<\text{Aik}&&>==\text{true} \), otherwise \( \text{is_constructible_v}<\text{Aik}, \text{cv_t}<\text{Aik}&&>==\text{true} \).

**TODO:** reword this paragraph

**Remarks:** The types \( \text{ctypes} \) shall be equal to the ordered sequence of the extended types \( \text{Args}_0,...,\text{Args}_1,...,\text{Args}_{n-1}... \)

where \( n \) is equal to \( \text{size_of}<\text{PTs}>(). \) Let \( ei \) be the \( i \) th ordered sequence of tuple elements of the resulting tuple object corresponding to the type sequence \( \text{Args}_i \).

**TODO:** reword this paragraph

**Returns:** A tuple object constructed by initializing the \( ki \) th type element \( eik \) in \( ei \)... with \( \text{get}<ki>(\text{std::forward}<\text{Ti}>(pti)) \) for each valid \( ki \) and each group \( ei \) in order.

**Note:** An implementation may support additional types in the parameter pack \( \text{Tuples} \) that support the tuple-like protocol, such as pair and array.

**Function Template** \( \text{product_type}::\text{make_from} \)

```cpp
template <class T, class PT>
constexpr "see below" make_from(PT&& pt);
```

**Effects:** Given the exposition-only function:

```cpp
template <class T, class PT, size_t... I>
constexpr T make_from_impl(PT&& t, index_sequence<I...}) { // exposition only
  return T(product_type::get-I<std::forward<Tuple>(t)...);
}
```

Equivalent to:

```cpp
return make_from_impl-T<std::forward<Tuple>(t),
                       product_type::element_sequence_for<PT>[]);
```
[Note: The type of T must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. - end note]

**Function Template**  \( \text{product\_type::move} \)

\[
\text{template } < \text{class PT1, class PT2>}
\]
\[
\text{PT1\& move(PT1\& pt1, PT2\&\& pt2)};
\]

In the following paragraphs, let \( \text{VPT2} \) be \( \text{remove\_cv}\_t<\text{remove\_reference}\_t<\text{PT2}> \), \( \text{Ti} \) be \( \text{product\_type::element\_t<i, PT1>} \) and \( \text{Ui} \) \( \text{product\_type::element\_t<i, VPT2>} \).

**Requires:** Both \( \text{PT1} \) and \( \text{VPT2} \) are ProductTypes with the same size, \( \text{product\_type::size<PT1>::value==product\_type::size<VPT2>::value} \) and \( \text{is\_assignable\_v<Ti, Ui\&\&> is true for all i.} \)

**Effects:** Moves each element of \( \text{pt2} \) to the corresponding element of \( \text{pt1} \).

**Function Template**  \( \text{product\_type::swap} \)

\[
\text{template } < \text{class PT>}
\]
\[
\text{void swap(PT\& x, PT\& y) noexcept('see below');}
\]

**Remark:** The expression inside \( \text{noexcept} \) is equivalent to the logical and of the following expressions: \( \text{is\_nothrow\_swappable\_v<Ti>} \) where \( \text{Ti} \) is \( \text{product\_type::element\_t<i, PT>} \).

**Requires:** Each element in \( \text{x} \) shall be swappable with (17.6.3.2) the corresponding element in \( \text{y} \).

**Effects:** Calls \( \text{swap} \) for each element in \( \text{x} \) and its corresponding element in \( \text{y} \).

**Throws:** Nothing unless one of the element-wise \( \text{swap} \) calls throws an exception.

**Function Template**  \( \text{product\_type::to\_tuple} \)

\[
\text{template } < \text{class PT>}
\]
\[
\text{constexpr 'see below' to\_tuple(PT\&\& pt)};
\]

**Effects:** Equivalent to ......

\[
\text{product\_type::make\_from\_tuple\_product\_type::element\_t<0, PT, ...>>(pt)};
\]

Change 20.5.1p1 [tuple.general], Header synopsis as indicated.

Replace

\[
\text{template <class... Tuples> constexpr tuple<CTypes...> tuple\_cat(Tuples&&... tpls)};
\]

by

\[
\text{template <class... PTs> constexpr tuple<CTypes...> tuple\_cat(PTs&&... pts)};
\]

Change 20.5.2 [tuple.tuple], class template tuple synopsis, as indicated.

Replace
Connor from a product type

**Suppress in 20.5.2.1p3, Construction [tuple.cnstr]**

Let \( \text{product_type::element}_i \) be the \( i \)th type in a template parameter pack named \( \text{UTypes} \), where indexing is zero-based.

**Replace 20.5.2.1p15-26, Construction [tuple.cnstr] by**

```
// 20.4.2.1, tuple construction
...
```

```
// 20.4.2.2, tuple assignment
...
```

```
// allocator-extended constructors
...
```

```
// 20.4.2.1, tuple construction
...
```

**Assignment from a product type**

**Suppress in 20.5.2.2p1, Assignment [tuple.assign]**
and \( \text{Ui} \) be the \( i \)th type in a template parameter pack named \( \text{UTypes} \), where indexing is zero-based.

Replace 20.5.2.2p9-20, Assignment [tuple.assign] by

```cpp
template <class PT>
tuple operator=(PT&& u);
```

Let \( \text{Ui} \) is \( \text{product_type::element_t<i, remove_cv_t<remove_reference_t<<PT>>>} \).

**Effects:** For all \( i \), assigns \( \text{std::forward<Ui>(product_type::get<i>(u))} \) to \( \text{product_type::get<i>(*this)} \)

**Returns:** *this

**Remarks:** This function shall not participate in overload resolution unless \( \text{PT} \) is a product type with the same number elements than this tuple and
\( \text{is_assignable<Ti&, const Ui&>::value} \) is true for all \( i \).

[Note: - We could as well say equivalent to \( \text{product_type::assign(std::forward<PT>(u), *this)} \); return *this. end note]

### Allocator-extended constructors from a product type

Change the signatures

```cpp
template <class Alloc, class PT>
EXPLICIT tuple(allocation_arg_t, const Alloc& a, const PT&);
```

by

```cpp
template <class Alloc, class PT>
EXPLICIT tuple(allocation_arg_t, const Alloc& a, PT&&);
```

**std::tuple_cat**

Adapt the definition of \( \text{std::tuple_cat} \) in [tuple.creation] to take care of product type

Replace \( \text{Tuples} \) by \( \text{PTs} \), \( \text{tpls} \) by \( \text{pts} \), \text{tuple} by product type, \text{get} by \text{product_type::get} and \text{tuple_size} by \text{product_type::size}.

```cpp
template <class... PTs>
constexpr tuple<Types...> tuple_cat(PTs&&... pts);
```

[Note: - We could as well say equivalent to \( \text{product_type::cat<tuple>(std::forward<PT>(pts)...)} \); end note]

**std::apply**

Adapt the definition of \( \text{std::apply} \) in [tuple.apply] to take care of product type

Replace \( \text{Tuple} \) by \( \text{PT} \), \text{t} by \( \text{pt} \), \text{tuple} by product type, \text{std::get} by \text{product_type::get} and \text{std::tuple_size} by \text{product_type::size}.

```cpp
template <class F, class PT>
constexpr decltype(auto) apply(F&& f, PT&& t);
```

[Note: - We could as well say equivalent to \( \text{product_type::apply(std::forward<F>(f), std::forward<PT>(t))} \); end note]

**std::pair**

Change 20.3.2 [pairs.pair], class template pair synopsis, as indicated:

Replace
template <class... Args1, class... Args2>
    pair<piecewise_construct_t, tuple<Args1...>> first_args, tuple<Args2...>> second_args);

by

template <class PT1, class PT2>
    pair<piecewise_construct_t, PT1 first_args, PT2 second_args>;

Add
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```cpp
template <class PT>
array& operator=(PT&& u);
```

Let \( \text{Ui} \) is `product_type::element_t<i, remove_cv_t<remove_reference_t<PT>>>`. 

Effects: For all \( i \) in \( 0..1 \), assigns `std::forward<Ui>(product_type::get<i>(u))` to `product_type::get<i>(*this)` 

Returns: `*this` 

Remarks: This function shall not participate in overload resolution unless `PT` is a product type with \( N \) elements and `is_assignable<T&, const Ui>::value` is true for all \( i \). 

[Note: - We could as well say equivalent to `product_type::assign(std::forward<PT>(u), *this); return *this` end note]

### Implementability

This is a library proposal. There is an implementation `PT_impl` of the basic ProductType algorithms. The standard library has not been adapted yet.

### Open Questions

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 the IS or a TS? If a TS which one? 
- Do we want to adapt `std::tuple_cat` 
- Do we want to adapt `std::apply` 
- Do we want the new constructors for `std::pair` and `std::tuple`

### Future work

Add other parts of the current standard if we have missed them.

### Acknowledgments

Thanks to all those that help on [P0327R1](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0327r1.pdf).

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

### References

- [N4387](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4387.html) Improving pair and tuple, revision 3
- [N4475](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4475.pdf) Default comparisons (R2)
- [P0095R1](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0095r1.pdf) Pattern Matching and Language Variants
- [P0327R1](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0327r1.pdf) Product Type Access (Revision 1)
- [P0327R2](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0327r2.pdf) Product Type Access (Revision 2)
- [P0338R0](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0338r0.pdf) C++ generic factories
- [P0343R0](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0343r0.html) Meta-programming High-Order Functions
- **PT.impl** Product types access emulation and algorithms
  

- **SWAPPABLE** ProductTypes must be Swappable by default
  
  https://github.com/viboes/std-make/tree/master/include/experimental/fundamental/v3/swappable

- **PT_SWAP** ProductTypes must be Swappable by default
  