Funkcyjność w C++

Krzysztof Pszeniczny

MIMUW

2016–01–20

Funkcyjność

Rysunek: xkcd

Haskell

```
f :: Num a => [a] -> a
f [] = 0
f (x : xs) = x * x + f xs

> f [1, 2, 3, 4] == 30
True
```

Listy w meta-C++

```
struct nil;
template<int Head, class Tail>
struct cons;

template<class List> struct F;
template<> struct F<nil> {
    static const int value = 0;
};
template<int x, class xs>
struct F<cons<x, xs> > {
    static const int value =
        x * x + F<xs>::value;
};

const int vals = F<
    cons<1, cons<2, cons<3, cons<4, nil> > > >
>::value;
```

meta-C++

```
template<int...> struct F;

template<> struct F<> {
    static constexpr auto value = 0;
};

template<int x, int... xs>
struct F<x, xs...> {
    static constexpr auto value =
        x * x + F<xs...>::value;
};

template<int... vals>
constexpr auto f = F<vals...>::value;

static_assert(f<1, 2, 3, 4> == 30, "???");
```

map – Haskell

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

map – meta-C++

```
template<
    template<class> class Fun,
    class List> struct map_t;

template<template<class> class Fun, class List>
using map = typename map_t<Fun, List>::type;

template<template<class> class Fun>
struct map_t<Fun, nil> {
    using type = nil;
};
```

map – meta-C++

```
template<
    template<class> class Fun,
    class Head, class Tail>
struct map_t<Fun, cons<Head, Tail>> {
    using type = cons<
        Fun<Head>,
        map<Fun, Tail>>;
};
```

map

```
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

map

```
template<class... Elems>
struct list {
    template<template<class...> class Fun>
    using map = list<Fun<Elems>...>;
};
```

map

```
template<class...> struct list;

template<> struct list<> {
    template<template<class...> class>
    using map = list<>;
};

template<class Head, class... Tail>
struct list<Head, Tail...> {
    template<template<class...> class Fun>
    using map = list<Fun<Head>, Fun<Tail>...>;
};
```

map

```
map :: (a -> b) -> [a] -> ([b] -> r) -> r
map f l k = k [f x | x <- l]
```

map

```
template<
    template<class...> class Fun,
    class... Args
    >
struct map {
    template<template<class...> class Cont>
    using then = Cont<Fun<Args>...>;
};

using example =
    map<std::vector, int, bool>
    ::then<std::vector>;
```

map

```
template<template<class...> class, class>
struct map_over_t;

template<
    template<class...> class Fun,
    template<class...> class Constructor,
    class... Args>
struct map_over_t<Fun, Constructor<Args...>> {
    using type = typename map<Fun, Args...>
        ::template then<Constructor>;
};

template<template<class...> class Fun, class Type>
using map_over =
    typename map_over_t<Fun, Type>::type;
```

map

```
static_assert(std::is_same<
    map_over<std::make_unsigned_t,
        std::pair<short, char>>,
    std::pair<unsigned short, unsigned char>
>{}, "???");
```

filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x : xs)
    | f x = x : filter f xs
    | otherwise = filter f xs
```

filter

```
template<class... Args>
struct list;

template<>
struct list<> {
    /* ... */

    template<class NewHead>
    using cons = list<NewHead>;

    template<template<class...> class>
    using filter = list<>;
};
```

filter

```
template<class Head, class... Tail>
struct list<Head, Tail...> {
    /* ... */

    using head = Head;
    using tail = list<Tail...>;

    template<class NewHead>
    using cons = list<NewHead, Head, Tail...>;

    template<template<class...> class Pred>
    using filter = typename filter_aux<Pred>::type;
};
```

filter

```
/* ... */
private:
template<
    template<class...> class Pred,
    bool=Pred<Head>::value>
struct filter_aux {
    using type = typename tail
        ::template filter<Pred>;
};

template<template<class...> class Pred>
struct filter_aux<Pred, true> {
    using type = typename tail
        ::template filter<Pred>
        ::template cons<Head>;
};
/* ... */
```

Przetestujmy...

```
static_assert(std::is_same<
    list<int, float, double, short>
        ::filter<std::is_integral>
        ::map<std::make_unsigned_t>
        ::map<std::vector>,
    list<std::vector<unsigned int>,
         std::vector<unsigned short>>
>{}, "???");
```

filter

```
filter :: (a -> Bool) -> [a] -> ([a] -> r) -> r
filter _ [] k = k []
filter f (x : xs) k
    | f x = filter f xs (\r -> k (x : r))
    | otherwise = filter f xs k
```

filter

```
template<template<class> class, class...>
struct filter;

template<template<class> class Predicate>
struct filter<Predicate> {
    template<template<class...> class Cont>
    using then = Cont<>;
};
```

filter

```
template<
    template<class> class Predicate,
    class Head,
    class... Tail>
struct filter<Pred, Head, Tail...> {
    /* ... */

    template<template<class...> class Cont>
    using then = typename
        then_t<Predicate<Head>::value, Cont>
        ::type;
};
```

filter

```
/* ... */
    template<
        bool WithHead,
        template<class...> class Cont>
    struct then_t {
        using type = typename filter<Pred, Tail...>
            ::template then<Cont>;
    };

    template<template<class...> class Cont>
    struct then_t<true, Cont> {
        template<class... Args>
        using new_cont = Cont<Head, Args...>;
        using type = typename filter<Pred, Tail...>
            ::template then<new_cont>;
    };
/* ... */
```

Przetestujmy...

```
static_assert(std::is_same<
        filter<std::is_integral,
            int, float, void, long long,
            int[], short
        >::then<std::tuple>,
        std::tuple<int, long long, short>
    >{}, "???");
```

Fold

```
template<>
struct list<> {
    /* ... */
    template<class Init,
        template<class, class...> class Merger>
    using foldl = Init;

    template<class Init,
        template<class, class...> class Merger>
    using foldr = Init;
};
```

Fold

```
template<class Head, class... Tail>
struct list<Head, Tail...> {
    /* ... */
    template<class Init,
        template<class, class...> class Merger>
    using foldl = typename tail
        ::template foldl<Merger<Init, Head>, Merger>;

    template<class Init,
        template<class, class...> class Merger>
    using foldr = Merger<Head,
        typename tail::template foldr<Init, Merger>>;
};
```

Częściowa aplikacja

```
template<
    template<class...> class Fun,
    class... Args>
struct apply {
    template<class... Rest>
    using type = Fun<Args..., Rest...>;
};

using example =
    apply<std::tuple, int, bool>
    ::type<double, int>;
```

Składanie

```
template<
    template<class...> class F,
    template<class...> class G>
struct compose {
    template<class... Args>
    using type = F<G<Args>>;
};
```

Monada Maybe

```
struct nothing {
    template<template<class...> class>
    using map = nothing;

    template<template<class...> class>
    using bind = nothing;
};

template<class T>
struct just {
    template<template<class...> class F>
    using map = just<F<T>>;

    template<template<class...> class F>
    using bind = F<T>;
};
```

Przykład

```
template<class T>
struct unvector_t {
    using type = nothing;
};

template<class T>
struct unvector_t<std::vector<T>> {
    using type = just<T>;
};

template<class T>
using unvector = typename unvector_t<T>::type;
```

Przykład

```
template<class T>
using operation = typename just<T>
    ::template bind<unvector>
    ::template bind<unvector>
    ::template bind<unvector>
    ::template map<std::make_unsigned_t>;

using input =
    std::vector<std::vector<std::vector<int>>>;

static_assert(
    std::is_same<operation<int>, nothing>{},
"???");
static_assert(
    std::is_same<operation<input>, just<unsigned>>{},
"???");
```

Monada State

```
/* We want: */
struct action {
    template<class S>
    struct run {
        using state = /* ... */;
        using value = /* ... */;
    };
};
```

CRTP

```
template<class Self>
struct state {
    /* map, bind, ... */
};
```

Monada State

```
template<class T>
struct pure : state<pure<T>> {
    template<class S>
    struct run {
        using state = S;
        using value = T;
    };
};
```

Monada State

```
struct get : state<get> {
    template<class S>
    struct run {
        using state = S;
        using value = S;
    };
};
```

Monada State

```
template<class T>
struct put : state<put<T>> {
    template<class S>
    struct run {
        using state = T;
        using value = void;
    };
};
```

Monada State

```
template<template<class...> class Fun>
using modify = typename get
    ::template map<Fun>
    ::template bind<put>;
```

Monada State

```
template<class Self>
struct state {
    template<template<class...> class Fun>
    struct map : state<map<Fun>> {
        template<class S>
        struct run {
            private:
            using self = typename Self
                ::template run<S>;

            public:
            using state = typename self::state;
            using value = Fun<typename self::value>;
        };
    };

    /* ... */
};
```

Monada State

```
/* ... */
    template<template<class...> class Then>
    struct bind : state<bind<Then>> {
        template<class S>
        struct run {
            private:
            using self = typename Self
                ::template run<S>;
            using next =
                typename Then<typename self::value>
                ::template run<typename self::state>;

            public:
            using state = typename next::state;
            using value = typename next::value;
        };
    };
/* ... */
```

Monada State

```
template<class T>
struct constant {
    template<class...>
    using type = T;
};

/* ... */
    template<class Then>
    using then = bind<constant<Then>::template type>;
/* ... */
```

Monada State

```
template<int N> struct proxy;

template<class> struct inc_t;

template<int N>
struct inc_t<proxy<N>> {
    using type = proxy<N + 1>;
};

template<class T>
using inc = typename inc_t<T>::type;
```

Monada State

```
using plus3 = modify<inc>
    ::then<modify<inc>>
    ::then<modify<inc>>;

using operation = plus3
    ::then<plus3>
    ::then<pure<int>>
    ::map<std::make_unsigned_t>;

using run = operation::run<proxy<42>>;

static_assert(std::is_same<run::state, proxy<48>>{},
    "???");
static_assert(std::is_same<run::value, unsigned>{},
    "???");
```

Krótkie definicje

```
template<class X, class Y>
using then = typename X::template then<Y>;

template<class List>
using sequence = typename List
    ::template foldr<pure<void>, then>;

template<template<class...> class Fun, class List>
using mapM = sequence<
    typename List::template map<Fun>
>;
```

Krótkie definicje

```
template<int n, class Action>
struct replicateM_t;

template<int n, class Action>
using replicateM = typename replicateM_t<n, Action>
    ::type;

template<int n, class Action>
struct replicateM_t {
    using type = typename Action
        ::template then<replicateM<n - 1, Action>>;
};

template<class Action>
struct replicateM_t<0, Action> {
    using type = pure<void>;
};
```

future

```
std::future<int> f = std::async(big_function);
/* ... */
int value = f.get();
```

N3721

- ```
  template<typename F>
  auto then(F&& func) ->
  future<decltype<func(*this)>>;
  ```
  ```

N3721

- `template<typename F> auto then(F&& func) -> future<decltype<func(*this)>>;`
- `template<typename T> future<typename decay<T>::type> make_ready_future(T&& value);`

N3721

- ```
  template<typename F>
  auto then(F&& func) ->
  future<decltype<func(*this)>>;
  ```
- ```
  template<typename T> future<typename
  decay<T>::type> make_ready_future(T&& value);
  ```
- ```
  future(future<future<R>>&& rhs) noexcept;
  ```
  ```

N3721

- `template<typename F>`
  `auto then(F&& func) ->`
  `future<decltype<func(*this)>>;`
- `template<typename T> future<typename`
  `decay<T>::type> make_ready_future(T&& value);`
- `future(future<future<R>>&& rhs) noexcept;`
- `template<typename R2>`
  `future<R2> unwrap();` dla `R` będącego `future<R2>`

Konstruktor typu

- Takie `X`, że `X<T>` ma sens dla każdego (sensownego) typu `T`.

Konstruktor typu

- Takie `X`, że `X<T>` ma sens dla każdego (sensownego) typu `T`.
- Przykłady:

Konstruktor typu

- Takie `X`, że `X<T>` ma sens dla każdego (sensownego) typu `T`.
- Przykłady:
  - `std::future`

Konstruktor typu

- Takie `X`, że `X<T>` ma sens dla każdego (sensownego) typu `T`.
- Przykłady:
  - `std::future`
  - `std::vector`

Konstruktor typu

- Takie `X`, że `X<T>` ma sens dla każdego (sensownego) typu `T`.
- Przykłady:
  - `std::future`
  - `std::vector`
  - `std::experimental::optional`

Konstruktor typu

- Takie `X`, że `X<T>` ma sens dla każdego (sensownego) typu `T`.
- Przykłady:
    - `std::future`
    - `std::vector`
    - `std::experimental::optional`
    - `std::unique_ptr`

Konstruktor typu

- Takie `X`, że `X<T>` ma sens dla każdego (sensownego) typu `T`.
- Przykłady:
  - `std::future`
  - `std::vector`
  - `std::experimental::optional`
  - `std::unique_ptr`
  - `*`

Funktorialność

- Jeśli mam `X<T> x` oraz `F f`, to mogę zrobić `X<std::result_of_t<F(T)> map(f, x)`.

Funktorialność

- Jeśli mam X<T> x oraz F f, to mogę zrobić X<std::result_of_t<F(T)> map(f, x).
- Przy czym map(f, map(g, x)) = map(bind(f(g(_1)), x)

Funktorialność

- Jeśli mam `X<T> x` oraz `F f`, to mogę zrobić `X<std::result_of_t<F(T)> map(f, x)`.
- Przy czym `map(f, map(g, x)) = map(bind(f(g(_1)), x)`
- Oraz `map([](auto e) { return e; }, x) == x`

Monady

- `X` – funktor.

Monady

- `X` – funktor.
- Jeśli mam `T`, to mogę zrobić `X<T>`.

Monady

- `X` – funktor.
- Jeśli mam `T`, to mogę zrobić `X<T>`.
- Jeśli mam `X<X<T>>` to mogę zrobić `X<T>`.

Monady

- X – funktor.
- Jeśli mam T, to mogę zrobić X<T>.
- Jeśli mam X<X<T>> to mogę zrobić X<T>.
- I pewne prawa. . .

Monady

- `X` – funktor.
- Jeśli mam `T`, to mogę zrobić `X<T>`.
- Jeśli mam `X<X<T>>` to mogę zrobić `X<T>`.
- I pewne prawa. . .
- Z tego wynika: jeśli mam `X<T>` oraz `function< X<U>(T) >`, to mogę zrobić `X<U>`.