This slide intentionally left blank

There is nothing to read here

Обработка коллекций наизнанку: много функций, один аргумент

Вадим Винник

Мотивация

- Парадигма функционального программирования проникает в C++:
  - operator(), функциональные объекты;
  - оптимизация хвостовой рекурсии;
  - λ-функции;
  - std::function, std::bind;
  - std::transform, std::accumulate - аналоги fmap и foldl/foldr;
  - template metaprogramming - позволяет моделировать классы типов;
- Не просто пополнение новыми инструментами, а парадигмальный сдвиг!
- Понятия и методы ФП остаются малознакомыми широкому сообществу.
- Цель доклада - продемонстрировать логику программирования, тривиальную для ФП, но безумную с точки зрения традиций ООП.
- Показать, сколь проста её реализация на языке C++.

Немного программологии: интенсиональность

- Сущности в программировании выступают в трёх ролях:
  - просто сущности;
  - средства **построения** сущностей;
  - средства **применения** средств построения (в частности, средства **управления** средствами построения).
- Эта не имманентная характеристика сущности, а внешняя.
- Определяется не сущностью самой по себе, а способом её использования в конкретном контексте.
- Сущности в программировании - **интенсиональны**.
- Пример - функция:
  - чаще всего выступает средством построения объектов данных;
  - но в может и сама выступать объектом построения.

Немного программологии: композиционность

- Композиции строят более сложные программные сущности из относительно простых.
  - Чаще всего имеют в виду композиции программ, но не менее важны композиции данных, данных с программами и т.д.
- Обычно их относят к интенсиональному типу средств построения.
- Не менее важно рассматривать их и как средства применения и, в частности, управления применением:
  - циклическая композиция управляет выполнением оператора-тела, который выступает средством построения объектов данных;
  - объект, реализующий идиому RAII, управляет применением заданного действия.
- Композицию можно даже относить к интенсиональному типу 1 (гомоиконичность, LISP), но это далеко уводит от темы доклада.

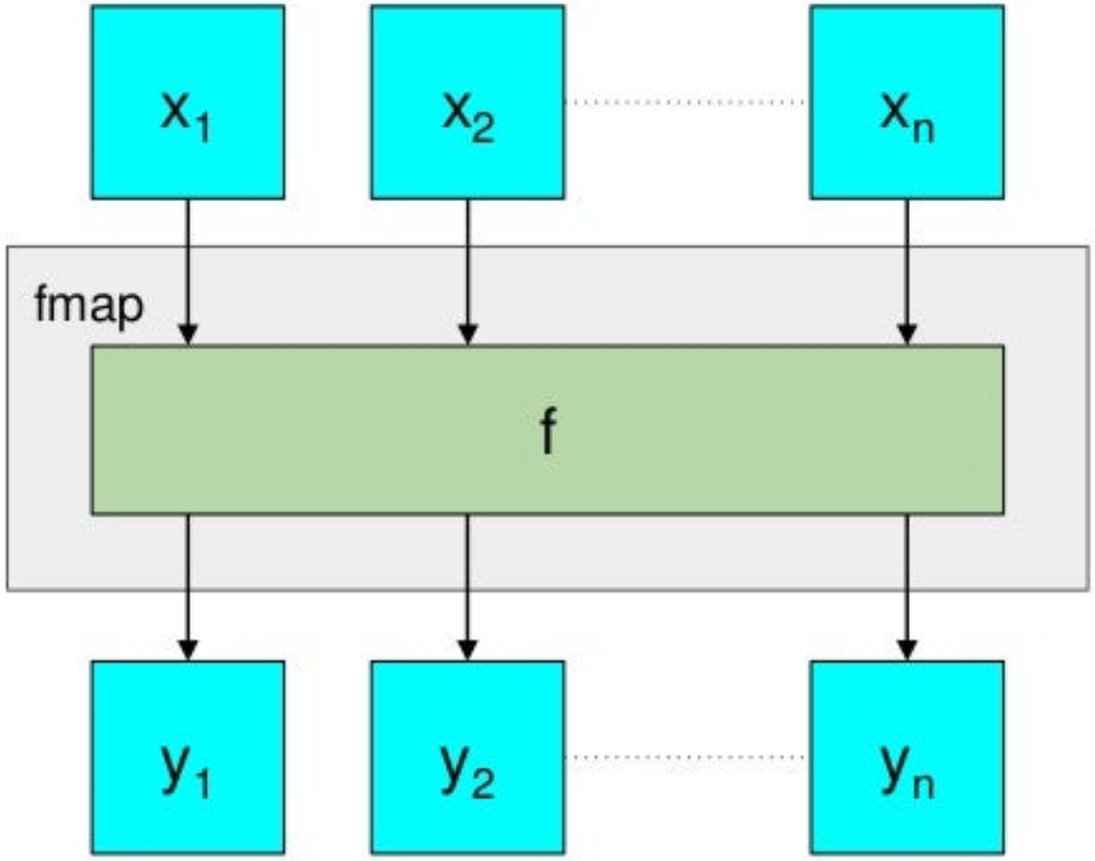
Немного программологии: композиционность

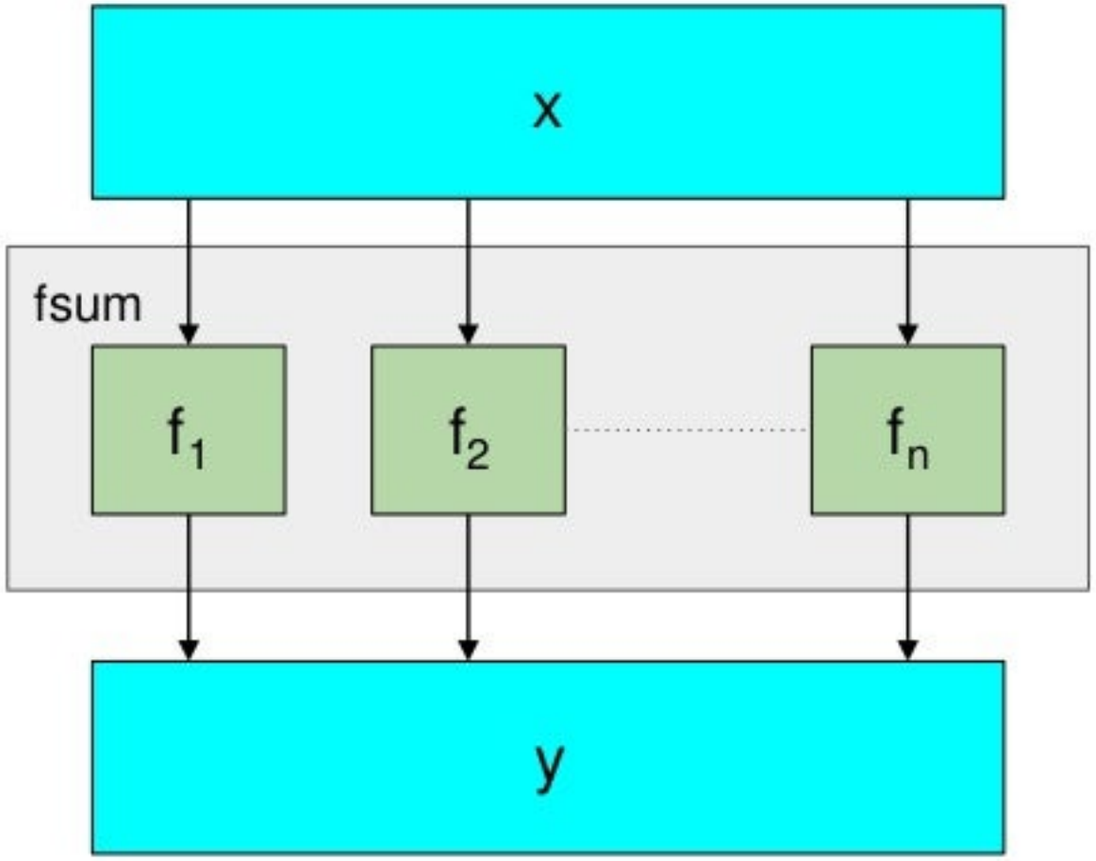
- Всякое ответвление программирования характеризуется своим набором композиций.
- Эти композиции:
  - выступают инструментами сочленения более сложных единиц из простых;
  - управляют применением частей в составе целого в процессе его работы.
- Именно композиции определяют суть и дух всякого программирования.
- Суть функционального программирования - не в чистоте функций и отказе от разрушающего присваивания.
- Напротив: чистота функций - средство для поддержки композиций ФП.

Проникновение ФП в C++

- Импорт композиций ФП в среду языка C++ и сложившейся вокруг него культуры программирования.
- Композиции, нашедшие широкое применение в ФП, для человека, привычного к ООП, могут выглядеть странно и пугающе.
- Композиции ФП часто основаны на λ-исчислении, комбинаторах и теории категорий (функторы, монады).
- **Обобщённая цель** доклада: показать, как выгод, которые в ФП достигаются с помощью монад, можно добиться штатными средствами языка C++, не упоминая монады в явном виде.

Вместо тысячи слов





Пример

- Пусть дана совокупность функций каждая из которых может дать искомый результат или **неудачу**.
- Например:
  - Попытаться извлечь параметр hostname из файла настроек в текущей директории;
  - Попытаться извлечь hostname из настроек в домашней директории пользователя;
  - Попытаться извлечь hostname из глобального файла настроек;
  - Взять hostname по умолчанию, прошитый в программе.
- Каждая, кроме последней, может окончиться неудачей: отсутствует файл или в файле нет такого параметра.
- Чтобы получить правильный hostname, нужно пытаться применять эти функции **в том же порядке** одну за другой **до первой удачи**.

Немного обозначений для краткости

- Если T - тип, то вместо std::optional<T> будем писать $T^?$ (**возможно**, T).
- Специальный объект “отсутствие” (std::nullopt) обозначим $\perp$ (**дно**).
- Рассматриваем функции типа $A_1,\ldots, A_m \to T^?$.
- Функция $\perp$ (**пустая**) - для любых кортежей x правильного типа $\perp(x) = \perp$.
- **Всюду определённые** функции: $t(x) \neq \perp$ для любых x (Tot).
- Отношение **порядка**: $f \preccurlyeq g$ (“f есть **подфункция** g”), если для любых x из $f(x) \neq \perp$ следует $g(x) = f(x)$.
- Функция $\perp$ - **наименьшая**, а любая всюду определённая t - **максимальная** (*но не наибольшая*) относительно порядка $\preccurlyeq$.
- Одноточечная функция: $p^{[a,\ y]}(x) = y$, если $x = a$; иначе $p^{[a,\ y]}(x) = \perp$.
- **Совместимость**: $f \approx g$, если из $f(x) \neq g(x)$ следует $f(x) = \perp$ или $g(x) = \perp$.

Сумма функций: определение

- Пусть дана совокупность функций $f_1,\ldots, f_n$ типа $A_1,\ldots, A_m \to T^?$.
- Их сумма $h = \sum[f_1,\ldots, f_n] = f_1 \oplus\ldots\oplus f_n$ есть функция того же типа.
- Если $f_k(x) = \bot$ для всех $k \in \{1, \ldots, n\}$, то $h(x) = \bot$.
- Иначе $h(x) = f_k(x)$, где $k \in \{1, \ldots, n\}$ - наименьшее такое число, что
  - $f_k(x) \neq \bot$,
  - $f_i(x) = \bot$ для всех $i \in \{1, \ldots, k-1\}$.
- Неформально говоря: сумма функций по очереди пробует применять все функции-слагаемые до первой удачи.
- Сумма функций терпит неудачу, если неудачу терпят все слагаемые.

Свойства суммы функций

- Некоммутативность: $(p^{[a, y]} \oplus p^{[a, z]})(a) = y$, но $(p^{[a, z]} \oplus p^{[a, y]})(a) = z$.
- Коммутативность при условии: если $f \approx g$, то $f \oplus g = g \oplus f$.
- Ассоциативность: $(f \oplus g) \oplus h = f \oplus (g \oplus h)$.
- Левый нуль: $\bot \oplus f = f$.
- Правый нуль: $f \oplus \bot = f$.
- Идемпотентность: $f \oplus g \oplus g = f \oplus g$
  - в программировании работает только для чистых функций.
- Неподвижность максимума слева: если $t \in Tot$, то $t \oplus f = t$.
- Сохранение максимума справа: если $t \in Tot$, то $(f \oplus t) \in Tot$.
- Монотонность: $f \preccurlyeq f \oplus g$.
- Поглощение меньшего большим: если $f \preccurlyeq g$, то $f \oplus g = g$.

Рекуррентное определение суммы функций

- База (сумма из 0 слагаемых есть 0):
  $\sum [\,] = \bot$.
- Шаг: выразить сумму n слагаемого через сумму n-1 слагаемых (n > 0):
  $h = \sum [f_1, \ldots, f_n]$ - это такая функция, что для любого x:
- Если $f_1(x) = y \neq \bot$, то $h(x) = y$.
- В противном случае $h(x) = g(x)$, где
  $g = \sum [f_2, \ldots, f_n]$.

Реализация

https://github.com/vadimvinnik/xfunctional

```
template <typename R, typename ...Args>
struct default_constf {
    using value_t = R;
    static R make(Args...) { return {}; }
};
```

Вспомогательные функции-константы

```
template <typename R, typename ...Args>
struct constf {
    using value_t = R;
    template <R Value>
    static constexpr R make(Args...) { return Value; }
};
```

создаются на этапе компиляции

```
template <typename R, typename ...Args>
class constf_t {
    R const value_;
public:
    using value_t = R;
    explicit constf_t(R const& value) : value_(value) {}
    R operator()(Args const&...) const { return value_; }
};
```

создаются на этапе выполнения

```
template <typename R, typename ...Args>
class single_point_t {
    R const value_;
    std::tuple<Args...> const args_;

public:
    using value_t = R;
    using maybe_t = std::optional<R>;

    single_point_t(R const& value, Args const&... args) :
    value_(value),
    args_(args...)
    {}

    maybe_t operator() (Args ...args) const {
        return args_ == std::make_tuple(args...)
            ? maybe_t {value_}
            : std::nullopt;
    }
};
```

одноточечная функция

$p^{[a,\, y]}$

```
template <typename R, typename ...Args>
struct fsum {
    using maybe_t = std::optional<R>;
    using funcptr_t = maybe_t(*)(Args...);

    static funcptr_t make() {
        return &(default_constf<maybe_t, Args...>::make);
    }

    template <typename F, typename ...Fs>
    static auto make(F f, Fs ...fs) {
        return [f, fs...](Args ...args) -> maybe_t {
            maybe_t const r = f(args...);
            return r.has_value()
                ? r
                : make(fs...)(args...);
        };
    }
```

строит сумму функций на этапе компиляции

прямая реализация рекуррентного определения

```
// continued:
// template <typename R, typename ...Args>
// struct fsum {

    template <typename I>
    static maybe_t exec(I from, I to, Args ...args) {
        while (from != to) {
        maybe_t const r = (*from)(args...);

            if (r.has_value())
                return r;

            ++from;
        }

        return std::nullopt;
    }
};
```

вычисление значения суммы функций на этапе выполнения

```
using sum_t = xfunctional::fsum<number_t, std::string>;

auto string_to_number = sum_t::make(
    decimal_to_number,
    english_numeral_to_number,
    roman_to_number);

assert(string_to_number("2019")   == 2019);
assert(string_to_number("twelve") == 12);
assert(string_to_number("XIV")    == 14);
assert(string_to_number("sieben") == std::nullopt);
```

пример использования

```
using sum_t = xfunctional::fsum<int>;

int count = 0;
auto chain = sum_t::make(
    [&count]() { ++count; return std::nullopt; },
    [&count]() { ++count; return std::nullopt; },
    [&count]() { ++count; return std::nullopt; },
    [&count]() { ++count; return 7;            },
    [&count]() { ++count; return std::nullopt; },
    [&count]() { ++count; return 3;            }
);

auto const value = chain();

assert(value == 7);
assert(count == 4);
```

демонстрация хода вычислений

```
using func_t = xfunctional::single_point_t<std::string, int>;
using sum_t = xfunctional::fsum<std::string, int>;

auto int_to_string = sum_t::make(
    func_t { "zero" , 0 },
    func_t { "one"  , 1 },
    func_t { "two"  , 2 },
    func_t { "zero2", 0 }, // !
    func_t { "three", 3 }
);

assert(int_to_string(0) == "zero");
assert(int_to_string(2) == "two");
assert(int_to_string(3) == "three");
assert(int_to_string(4) == std::nullopt);
```

окончание по первому успеху

Итоги и дальнейшие вопросы

- Связь с **шаблонами** “Chain of Responsibility” и “Composite”.
- Этот код работает на **production**.
- Для ряда практических задач стиль ФП - наиболее адекватное средство выражения их естественной логики.
- Если функция f имеет побочные эффекты, то бесконечная сумма ∑ [f, f, …] представляет собой… что?
- *Предел бесконечной монотонной последовательности* и заодно *неподвижная точка* операции ⊕f - это **цикл**!
- А теперь вывернем наизнанку! Применять следующую функцию к результату предыдущей **до первой неудачи**.
- Получим **монаду** Maybe - но это уже совсем другая история.

```
} // the talk
```