Обработка коллекций в функциональном стиле: от рекурсий к гиломорфизмам

Вадим Винник

Об авторе

- SolarWinds MSP.
- Technical lead developer.
- C++, C# for work.
- Haskell just for fun.
- Domains:
  - Network traffic filtering.
  - Data backup and recovery.
  - Business process simulation.
- Ph.D., lecturer in universities.
- vadym.vinnyk@solarwinds.com.
- vadim.vinnik@gmail.com.

Предпосылки

- Язык C++, эволюционируя, приобретает черты функционального.
  - Вспомните вчерашний мастер-класс.
- Функциональное программирование набирает популярность.
  - Перестаёт быть лабораторной диковинкой.
- В функциональных языках хорошо развиты средства обработки списков.
  - LISP = LISt Processing.
  - Три кита ФП: чистые функции, алгебраические типы данных и категория типов.
- Ряд средств для обработки последовательностей в функциональном стиле уже есть:
  - Boost Ranges, range-v3, think-cell range.
- Но остаётся одна недостаточно хорошо покрытая область...

Обработка коллекций

- Элементы - однотипны.
  - Полиморфизм не запрещён…
  - но скрыт за единым для всех элементов интерфейсом.
- Элементы образуют линейную последовательность.
  - На уровне внутренней реализации структура данных может быть любой.
  - Однако на уровне интерфейса коллекция выглядит как последовательность.
- Последовательность - не обязательно конечна.
  - В ходе выполнения программы обрабатывается конечное число элементов.
  - Однако оно может не быть ограничено заранее.
  - Бесконечные списки характерны для ленивых языков, особенно - функциональных.
  - Коллекция не хранит элементы, а генерирует их по мере необходимости.
- Однократный поэлементный проход по коллекции.

Примеры

- Отыскать наибольший, наименьший элемент.
- Вычислить сумму элементов.
- Пропустить заданное число элементов.
- Отсечь после заданного числа элементов.
- Обрыв последовательности по условию.
- Отфильтровать элементы, удовлетворяющие условию.
- Преобразовать элементы (map, transform).

Как обработать коллекцию

- Цикл for с целочисленным индексом
  - ниже всякой критики и недостойно упоминания здесь;
- Цикл for с итераторами
  - ниже границы допустимого;
- Цикл for по диапазону (range-based for)
  - просто плохо в большинстве случаев;
- Высокоуровневые алгоритмы из std:: (copy, all_of, find_if)
  - нормально
- Свёртки
  - полностью универсально, но, пока не столь хорошо знакомо широким массам…
    - для того этот доклад и предназначен!

Свёртка: начальный уровень

- Сумма последовательности чисел $s = \sum\{a_1,\ldots,a_n\}$.
- Рекуррентное определение:
  - $s_0 = 0$;
  - $s_{k+1} = s_k + a_{k+1}$.
- Пример: $s_3 = s_2+a_3 = (s_1+a_2)+a_3 =(((s_0+a_1))+a_2)+a_3 =(((0+a_1))+a_2)+a_3$.
- Это - левая свёртка по операции “+” с начальным значением 0.
- Правая свёртка - симметрично: $a_1+(a_2+(a_3+0)))$.
- Существует std::accumulate
  - Определена в #include <numeric>, явно неудачно.
  - Гораздо мощнее и универсальнее, чем принято думать.
  - Лишь очень частный случай из стройной системы понятий, разработанных в ФП.

Сведение различных алгоритмов к свёрткам

- Наибольший элемент последовательности целых:
  - свёртка по функции `std::max<int>`
  - с начальным значением `std::numeric_limits<int>::min()`.
- Удовлетворяют ли все элементы типа T предикату p:
  - свёртка по `[p](bool b, T x) { return b && p(x); }`,
  - начальное значением `true`.
- Переворачивание контейнера:
  - свёртка по `[](std::list<T> x, T y) { x.push_front(y); return x; }`,
  - начальное значение `std::list<T>()`.
- Применение последовательности функций, $f_n(...f_2(f_1(x))...)$:
  - свёртка по `[](T x, std::function<T(T)> f) { return f(x); }`,
  - начальное значение - x.

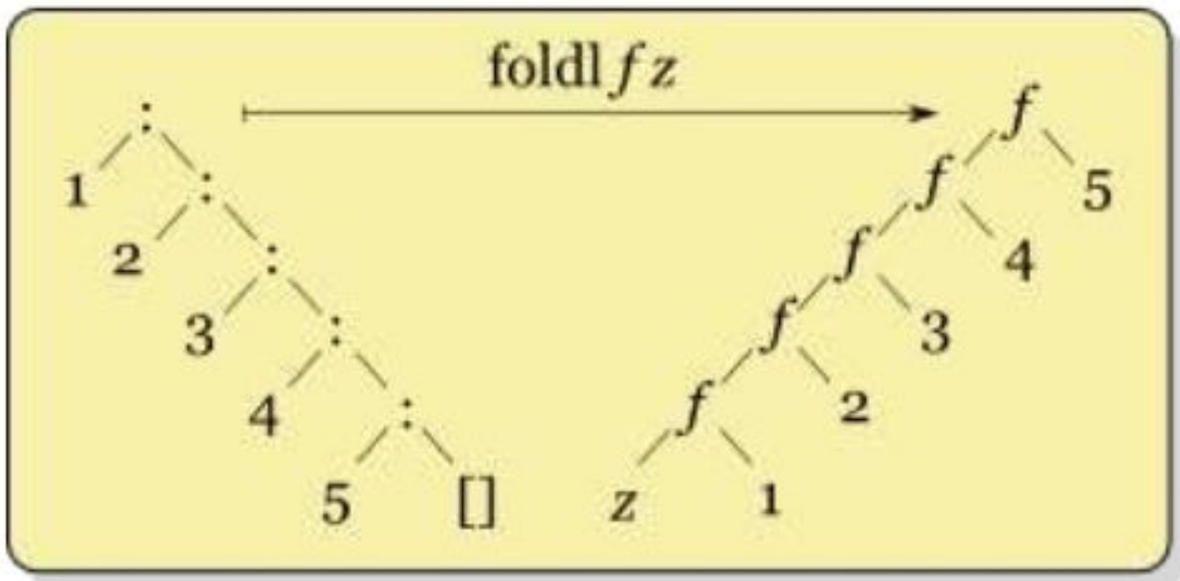
Свёртка, общее определение (псевдокод)

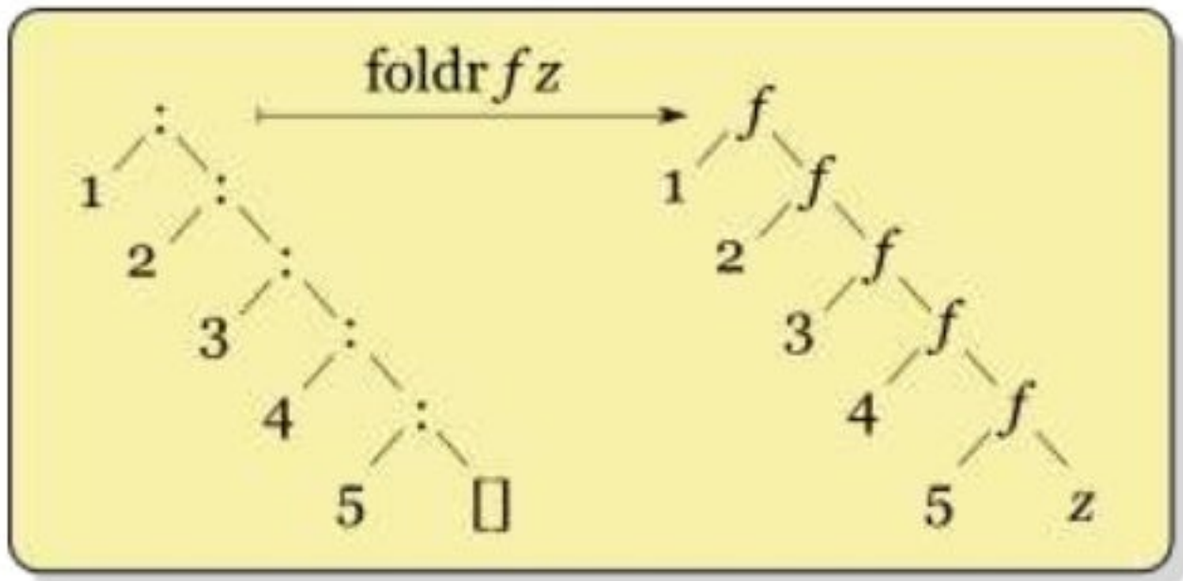
- f : U×T → U.
- a ∈ U.
- (foldl f a) : [T] → U.
- (foldl f a) [] = a.
- (foldl f a) (t:ts) = (foldl f (f a t)) ts.
- foldl : (U×T→U) × U → ([T]→U)

- g : T×U → U.
- a ∈ U.
- (foldr g a) : [T] → U.
- (foldr g a) [] = a.
- (foldr g a) (t:ts) = g t (foldr g a) ts.
- foldr : (T×U→U) × U → ([T]→U).

- Функция высшего порядка: аргумент и значение суть функции.
- Математически - совершенно симметричны.
- Одинаково ли удобны для программной реализации через рекурсию?

Левая и правая свёртки





- В случае левой свёртки - хвостовая рекурсия.
- Хорошо оптимизируется.

Возможная реализация правой свёртки

```
template <typename Result, typename Func, typename FwIt>
Result foldr(Func func, FwIt const from, FwIt const to, Result const seed)
{
  if (from == to)
    return seed;
  auto const item = *from;
  auto const next = advance(from); // replacement for from+1 and ++from
  auto const rest = foldr(func, next, to, seed);
  return func(item, rest);
}
```

Почти эквивалентно

```
return from == to
        ? seed
        : func(
                *from,
                foldr(
                        func,
                advance(from),
                to,
                seed));
```

Возможная реализация левой свёртки

```
template <typename Result, typename Func, typename FwIt>
Result foldl(Func func, FwIt const from, FwIt const to, Result const seed)
{
  if (from == to)
    return seed;
  auto const new_seed = func(*from, seed);
  auto const next = advance(from);
  return foldl(func, next, to, new_seed);
}
```

Почти эквивалентно

```
return from == to
        ? seed
        : foldl(
                func,
                advance(from),
                to,
        func(
                *from,
                seed));
```

Использование

```
std::string append_int(int x, std::string const& s) {
    return s + std::to_string(x) + " ";
}

std::list<int> data = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

std::cout
        << foldl<std::string>(append_int, std::begin(data), std::end(data), "")
        << std::endl  // 0 1 2 3 4 5 6 7 8 9
        << foldr<std::string>(append_int, std::begin(data), std::end(data), "")
        << std::endl; // 9 8 7 6 5 4 3 2 1 0
```

Обобщение свёртки - катаморфизм

- Предыдущий вариант работает для пары итераторов.
- А вдруг структура данных играет роль контейнера, но итераторами в стиле STL не обладает?
  - Самодельный связный список, например.
- Тогда функция обобщённой свёртки должна принимать:
  - состояние прохода по контейнеру;
  - функцию-распознаватель окончания прохода;
  - функцию-выделитель очередного элемента из состояния;
  - функцию-продвигатель состояния на следующий элемент.
- Альтернатива:
  - Конец прохода представлен отсутствием состояния (std::optional);
  - Распознавать конец прохода, выделять очередной элемент и переходить к следующему одним действием.

Возможная реализация

```
template <
    typename Result,
    typename Func,
    typename Unconser,
    typename State>
Result foldl(
    Func func,
    Unconser unconser,
    State state,
    Result const& seed)
{
```

```
auto const maybe_uncons = unconser(state);
return !maybe_uncons.has_value()
  ? seed
  : foldl(
      func,
      unconser,
      maybe_uncons.value().second,
      func(
        *maybe_uncons.value().first,
        seed));
}
```

Использование-1: пара итераторов как состояние

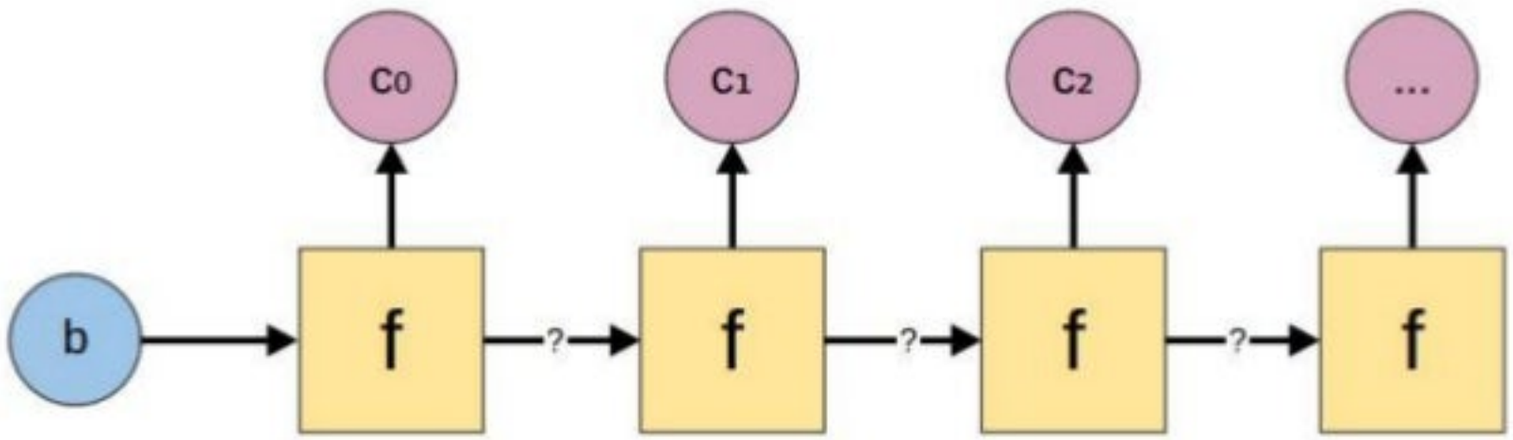
```
template <typename Container>
constexpr
container_range_t<Container> to_range(Container const& container)
noexcept
{
    return std::make_pair(std::cbegin(container), std::cend(container));
}
```

Использование-2: деконструкция диапазона

```
template <typename ForwardIt>
maybe_t<ForwardIt> uncons_range(range_t<ForwardIt> const range) {
        return range.first == range.second
                ? std::nullopt
                : std::make_optional(
                        std::make_pair(
                        range.first,
                        std::make_pair(
                                advance(range.first),
                                range.second)));
}
```

Операция, обратная к свёртке, - развёртка

- Поддержка в C++ не обнаружена.
- Зато есть в Haskell.
- Тип: `(b → Maybe (a, b)) → b → [a]`,
- Если (f x) == Nothing, то (unfoldr f x) == [],
- Иначе (f x) == Just (u, y), тогда (unfoldr f x) == u : (unfoldr f y).



Примеры развёрток

- Целое число n преобразовать в список
  - чисел [0, n);
  - разрядов двоичного или десятичного представления;
  - простых делителей;
  - чисел Фибоначчи до n-го включительно;
- Строку (длины n) преобразовать в список
  - символов;
  - лексем между заданными разделителями;
  - подстрок от символа номер k до конца строки, $k \in [0, n)$;
  - вариантов синтаксического разбора (согласно некоторой неоднозначной грамматики).

Гиломорфизмы

- Композиция развёртки (анаморфизма) и свёртки (катаморфизма).
- Развернуть первоначальное значение в контейнер $x \rightarrow [a_1, a_2, \ldots, a_n]$…
- …и свернуть заново, по иной операции: $[a_1, a_2, \ldots, a_n] \rightarrow y$.
- Может быть определён и как самостоятельное понятие. Даны:
  - типы A (исходный), B (промежуточный), C (результат);
  - константа $c \in C$;
  - предикат $p : A \rightarrow Boolean$ (условие окончания развёртки);
  - функция $f : A \rightarrow A \times B$ (развёртыватель);
  - функция $g: B \times C \rightarrow C$ (свёртыватель).
- Тогда гиломорфизм h есть функция типа $A \rightarrow C$.
  - Если p(a), то h(a) = c;
  - Иначе - пусть (a, b') = f(a), тогда h(a) = f(b, h(a')).

Как реализовать гиломорфизм на C++

```
template <
    typename Result,
    typename Unfolder,
    typename Combiner,
    typename Seed>
Result unfoldl(
    Unfolder unfolder,
    Combiner combiner,
    Result const& initial,
    Seed const& seed)
{
```

```
auto maybe_unfolded = unfolder(seed);
return !maybe_unfolded.has_value()
        ? initial
        : unfoldl(
            unfolder,
            combiner,
            combiner(
                    maybe_unfolded.value().first,
                    initial),
            maybe_unfolded.value().second);
}
```

Как реализовать развёртку на C++

- Гиломорфизм определяют как развёртку и последующую свёртку…
- Но мы сделаем наоборот!
- Подставим в гиломорфизм тривиальную свёртку, которая накапливает промежуточные значения b в контейнере.
- Тем самым, развёртка сведена к гиломорфизму.

Пример: генератор последовательности чисел

```
struct int_ascending_splitter {
    int_ascending_splitter(int _upper_bound): upper_bound(_upper_bound) {}

    std::optional<std::pair<int, int>> operator()(int x) const {
        return x < upper_bound
            ? std::make_optional(std::make_pair(x, x + 1))
            : std::nullopt;
    }

    int const upper_bound;
};
```

Пример: использование генератора

```
int_ascending_splitter generator(10);

auto s = unfoldl<std::string>(generator, append_int, "", 0);

// 0 1 2 3 4 5 6 7 8 9
```

Итоги и перспективы

- Подходы и идиомы, обычно характерные для языков функционального программирования, вполне можно реализовать и использовать в C++.
- Традиционно сильными сторонами функциональных языков считались гарантия корректности и лёгкость обработки сложно структурированных данных. Добро пожаловать в C++.
- Больше перегрузок богу перегрузок! Чтобы *морфизмами стало удобно пользоваться в ещё более разнообразных сценариях.
- Вопросы эффективности. Нужно добавить досрочный прерыватель свёртки. Сколь бы ни сближался язык C++ с функциональной парадигмой, вряд ли порядок вычисления станет ленивым.

```
}
```