Лекция 9б. Мемоизация и нестрогие вычисления

Коновалов А. В.

17 октября 2022 г.

Понятие мемоизации

Мемоизация — оптимизация, позволяющая избегать повторного вычисления функции, вызванной с теми же аргументами, как и в один из прошлых вызовов.

В общем случае нужно мемоизировать чистые функции: детерминированные и без побочных эффектов. Обоснование:

- Нет смысла мемоизировать функцию (`read port`) — она на каждом вызове должна выдавать очередное значение из файла.
- Нет смысла мемоизировать функцию (`display message`), т.к. она должна выполнять побочный эффект.

Мемоизация «нечистой» функции нужна очень редко

Пример мемоизации «нечистой» функции. Функция `(get-messages lang-id)` загружает из файла ресурсов сообщения программы на некотором языке. В процессе работы программы пользователь может залезть в настройки и поменять выбранный язык. Если функцию вызывать при каждой потребности вывести на экран сообщение, то она будет многократно читать один и тот же файл. Если загружать все ресурсы для всех языков в начале работы программы, то будут загружены лишние ресурсы. Приемлемый вариант — мемоизировать вызовы `get-messages`.

Приём мемоизации в Scheme (1)

Пусть нам нужно мемоизировать функцию вида

```
(define (func x y z)
  (+ x y z))
```

Данная функция должна при вызове с ранее известными аргументами «вспоминать» свой результат, не вычисляя его заново. Но если аргументы новые, она должна результат вычислить и запомнить его.

Приём мемоизации в Scheme (2)

Для этой цели нам нужно некоторое хранилище, где мы будем сопоставлять аргументы с вычисленными результатами. Для этой цели проще всего использовать ассоциативный список. Делать хранилище открытым тоже не стоит — снаружи должна быть видна только функция `func` (хороший стиль — минимизировать область видимости переменных). Соответственно, будем использовать приём статических переменных в языке Scheme:

```
(define func-memo
  (let ((known-results '()))
    (lambda (x y z)
        ... )))
```

Приём мемоизации в Scheme (3)

Если значения аргументов есть в хранилище, то нужно вернуть известный результат. Если нет — вычислить и положить в список `known-results`:

```
(define func-memo
  (let ((known-results '()))
    (lambda (x y z)
      (let* ((args (list x y z))
             (res (assoc args known-results)))
        (if res
            (cadr res)
            (let (res (+ x y z))
              (set! known-results
                    (cons (list args res) known-results))
              res))))))
```

Приём мемоизации в Scheme (4)

Так мы запоминаем все предыдущие значения. Иногда функция часто вызывается с теми же значениями, которые были в прошлый раз. Тогда имеет смысл запоминать только последнее значение:

```
(define func-memo-last
  (let ((last-arg #f)
        (known-result #f))
    (lambda (x y z)
      (let ((arg (list x y z)))
        (if (equal? arg last-arg)
          known-result
          (let ((res (+ x y z)))
            (set! last-arg arg)
            (set! known-result res)
            res))))))
```

Строгие и нестрогие вычисления (1)

Строгие вычисления — аргументы функции полностью вычисляются до того, как эта функция вызывается. Вызовы процедур в Scheme всегда строгие. Строгую стратегию вычислений часто называют call-by-value.

В случае **нестрогих вычислений** значения выражений могут вычисляться по необходимости, их вызов может быть отложен.

Примеры нестрогих вычислений:

- `(if cond then else)` — вычисляется либо `then`, либо `else`.
- `(and …)`, `(or …)`.
- В языке Си логические операции `&&`, `||` тоже не строгие.

Строгие и нестрогие вычисления (2)

В теории рассматривают две разновидности нестрогих вычислений:

- call-by-name, вызов по имени — нормальная редукция в лябмда-исчислении,
- call-by-need, вызов по необходимости — ленивые вычисления.

Строгие и нестрогие вычисления (3)

В некотором смысле разновидность call-by-name — макроподстановка:

```
(define-syntax double
  (syntax-rules ()
    ((double x) (+ x x))))

(define-syntax ++
  (syntax-rules ()
    ((++ var) (begin (set! var (+ var 1))
                     var))))

(define x 10)

(double (++ x))  ;; выведет 23
x                ;; выведет 12
```

Пример: вызов по имени в Алголе-60 (1)

```
function sum(i, first, last, val): real;
    integer i, first, last;
    real val;
    value first, last;
begin
  real res := 0;
  for i := first to last do
    res := res + val;

  sum := res
end;
```

Почему нельзя цикл заменить на `res := (end - start + 1) * val`?

Пример: вызов по имени в Алголе-60 (2)

```
function square(x): real;
    integer x;
begin
  square := x * x;
end;

real temperature[1 : 100];

integer k;

print(sum(k, 1, 10, square(k)));
print(sum(k, 1, 100, temperature[k]) / 100);
```

Стратегия call-by-need — Haskell

Пример стратегии `call-by-need` — ленивый язык программирования Хаскель

```
test xs = head (map (\x → x*x) xs)
```

Будет вычисляться квадрат только самого первого элемента списка.

Примитивы Scheme для обеспечения ленивых вычислений (1)

Это макрос `(delay expr)` и функция `(force promise)`. Макрос `delay` принимает выражение и формирует обещание (promise) вычислить это выражение, когда потребуется. `force` вычисляет этот promise, результат мемоизируется.

В первом приближении:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expr) (lambda () expr))))

(define (force promise)
  (promise))
```

Примитивы Scheme для обеспечения ленивых вычислений (2)

С мемоизацией:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expr) (list #f (lambda () expr)))))

(define (force promise)
  (if (car promise)
      (caar promise)
      (begin
        (set-car! (list ((cadr promise))))
        (caar promise))))
```