Функциональное программирование
Лекция 1. Лямбда-исчисление

Денис Николаевич Москвин

СПбАУ РАН, CS Center

12.02.2015

План лекции

1. Функциональное vs императивное программирование
2. Введение в λ-исчисление
3. Подстановка и преобразования
4. Расширения чистого λ-исчисления

План лекции

Императивное программирование

Императивное программирование: вычисление описывается в терминах инструкций, изменяющих состояние вычислителя.
В императивных программах:

- Состояние изменяется инструкциями присваивания: `v = E`
- Есть механизм условного исполнения: инструкции `if`, `switch`
- Есть механизм циклов: инструкции `while`, `for`
- Инструкции исполняются последовательно `C1; C2; C3`

Иногда говорят про стиль фон Неймана (Джон Бэкус)

Императивное программирование: пример

Скалярное произведение двух n-мерных векторов для фон-неймановского языка

```
res = 0;
for (i = 0; i < n; i++)
    res = res + a[i] * b[i];
```

Как здесь формируется состояние и какие изменения с ним происходят?

Императивное программирование: пример

Скалярное произведение двух n-мерных векторов для фон-неймановского языка

```
res = 0;
for (i = 0; i < n; i++)
    res = res + a[i] * b[i];
```

Как здесь формируется состояние и какие изменения с ним происходят?

Абстрактное представление (пока забываем про I/O):

$$\sigma_0 \Longrightarrow \sigma_1 \Longrightarrow \sigma_2 \Longrightarrow \ldots \Longrightarrow \sigma_n$$

Выполнение программы: переход *вычислителя* из начального *состояния* в конечное с помощью последовательных *инструкций*.

Функциональное программирование

Функциональная программа — *выражение*, её выполнение — вычисление (*редукция*) этого выражения.

- Нет состояний — нет переменных
- Нет переменных — нет присваивания
- Нет циклов, поскольку нет различий между итерациями
- Последовательность не важна, поскольку выражения независимы

Вместо этого есть:

- Рекурсия — вместо циклов
- Функции высших порядков (HOF)

Функциональное программирование: пример

Скалярное произведение для функционального языка

```
innerProduct = (sum .) . zipWith (*)
```

Абстрактное представление:

$$f_n(f_{n-1}(\ldots f_2(f_1(\sigma_0))\ldots))$$

Выполнение программы – *вычисление выражения*.

```
innerProduct as bs -> ((sum .) . zipWith (*)) as bs
                   -> (sum .) (zipWith (*) as) bs
                   -> (sum . zipWith (*) as) bs
                   -> sum (zipWith (*) as bs)
                   -> ...
```

Функциональное программирование: преимущества и недостатки

Преимущества ФП:

- Более «точная» семантика
- Большая свобода при исполнении (например, поддержка параллельности)
- Большая выразительность
- Лучшая параметризация и модульность
- Удобство при работе с «бесконечными» структурами данных

Недостатки ФП:

- Ввод-вывод и прочая интерактивность: нужен специальный инструментарий
- Быстродействие (исполнение в чуждой архитектуре)

Чистота и побочные эффекты

Императивный код:

```
bool flag;
int f (int n) {
    int retVal;
    if flag then retVal = 2 * n;
            else retVal = n;
    flag = ! flag;
    return retVal;
}
void test() {
    flag = true;
    printf("f(1) + f(2) = %d\n", f(1) + f(2));
    printf("f(2) + f(1) = %d\n", f(2) + f(1));
}
```

Каков будет вывод при вызове `test`?

Чистота и побочные эффекты

Императивный код:

```
bool flag;
int f (int n) {
    int retVal;
    if flag then retVal = 2 * n;
            else retVal = n;
    flag = ! flag;
    return retVal;
}
void test() {
    flag = true;
    printf("f(1) + f(2) = %d\n", f(1) + f(2));
    printf("f(2) + f(1) = %d\n", f(2) + f(1));
}
```

Каков будет вывод при вызове `test`?

```
f(1) + f(2) = 4
f(2) + f(1) = 5
```

Чистота и побочные эффекты (2)

- Функция `f` не является чистой (pure) математической функцией. Говорят о нарушении *ссылочной прозрачности (reference transparency)*.
- Результат вызова `f` зависит от внешних факторов, а исполнение приводит к возникновению побочного эффекта (side effect). Причина: глобально доступное состояние + разрушающее присваивание.
- В *чистом* функциональном программировании такие функции не используются (в Хаскелле живут в гетто `IO`, в других — допустимы, но не рекомендуются).

План лекции

λ-исчисление

- λ-исчисление — формальная система, лежащая в основе функционального программирования.
- Разработано Алонзо Чёрчем в 1930-х для формализации и анализа понятия вычислимости.
- Имеет бестиповую и множество типизированных версий.
- Дает возможность компактно описывать семантику вычислительных процессов.

Неформальное введение в λ-исчисление

- В λ-исчислении имеются два способа строить выражения:
  - *применение (аппликация, application)*;
  - *абстракция (abstraction)*.
- Нотация применения F к X:

$$F\,X$$

- С точки зрения программиста: F (алгоритм) применяется к X (входные данные).
- Однако явного различия между алгоритмами и данными нет, в частности допустимо самоприменение:

$$F\,F$$

Неформальное введение в λ-исчисление: абстракция

- Пусть $M \equiv M[x]$ — выражение, (возможно) содержащее x. Тогда абстракция

$$\lambda x.\, M$$

обозначает функцию

$$x \mapsto M[x],$$

то есть каждому x сопоставляется $M[x]$.

- Лямбда-абстракция — способ задать неименованную (анонимную) функцию.
- Если x в $M[x]$ отсутствует, то $\lambda x.\, M$ — константная функция со значением M.

Неформальное введение в λ-исчисление: $=_\beta$

- Применение и абстракция работают совместно:

$$\underbrace{(\lambda x.\, 2 \cdot x + 8)}_{F}\ \underbrace{17}_{X} \;=\; 2 \cdot 17 + 8 \quad (= 42).$$

- То есть $(\lambda x.\, 2 \cdot x + 8)\ 17$ — применение функции $x \mapsto 2 \cdot x + 8$ к аргументу 17, дающее в результате $2 \cdot 17 + 8$.
- В общем случае имеем β-эквивалентность

$$(\lambda x.\, M)\, N \;=_\beta\; M[x := N],$$

где $M[x := N]$ обозначает подстановку N вместо x в M.

Термы чистого λ-исчисления

Определение

Множество λ-***термов*** Λ индуктивно строится из переменных $V = \{x, y, z, \ldots\}$ с помощью применения и абстракции:

$$\begin{aligned} x \in V &\Rightarrow x \in \Lambda \\ M, N \in \Lambda &\Rightarrow (M\,N) \in \Lambda \\ M \in \Lambda, x \in V &\Rightarrow (\lambda x.\, M) \in \Lambda \end{aligned}$$

- В абстрактном синтаксисе

$$\Lambda \;::=\; V \mid (\Lambda\,\Lambda) \mid (\lambda V.\, \Lambda)$$

- **Соглашение.** Произвольные термы пишем заглавными буквами, переменные — строчными.

Термы (примеры)

Примеры λ-термов

$$x$$

$$(x\,z)$$

$$(\lambda x.\,(x\,z))$$

$$((\lambda x.\,(x\,z))\,y)$$

$$(\lambda y.\,((\lambda x.\,(x\,z))\,y))$$

$$((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w)$$

$$(\lambda z.\,(\lambda w.\,((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w)))$$

Термы (соглашения)

Общепринятыы следующие соглашения:

- Внешние скобки опускаются.
- Применение ассоциативно *влево*:

$$\mathrm{F\,X\,Y\,Z} \quad \text{обозначает} \quad (((\mathrm{F\,X})\,\mathrm{Y})\,\mathrm{Z})$$

- Абстракция ассоциативна *вправо*:

$$\lambda x\,y\,z.\,\mathrm{M} \quad \text{обозначает} \quad (\lambda x.\,(\lambda y.\,(\lambda z.\,(\mathrm{M}))))$$

- Тело абстракции простирается вправо насколько это возможно:

$$\lambda x.\,\mathrm{M\,N\,K} \quad \text{обозначает} \quad \lambda x.\,(\mathrm{M\,N\,K})$$

Термы (примеры)

Те же примеры, что и выше, но с использованием соглашений

$$
\begin{aligned}
x &= x \\
(x\,z) &= x\,z \\
(\lambda x.\,(x\,z)) &= \lambda x.\,x\,z \\
((\lambda x.\,(x\,z))\,y) &= (\lambda x.\,x\,z)\,y \\
(\lambda y.\,((\lambda x.\,(x\,z))\,y)) &= \lambda y.\,(\lambda x\,.\,x\,z)\,y \\
((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w) &= (\lambda y.\,(\lambda x\,.\,x\,z)\,y)\,w \\
(\lambda z.\,(\lambda w.\,((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w))) &= \lambda z\,w.\,(\lambda y.\,(\lambda x\,.\,x\,z)\,y)\,w
\end{aligned}
$$

Свободные и связанные переменные

Абстракция $\lambda x\,.\,M[x]$ *связывает* дотоле *свободную* переменную x в терме M.

Пример 1

$$(\lambda y.\,(\lambda x.\,x\,z)\,y)\,w$$

Переменные x и y — связанные, а z и w — свободные.

Пример 2

$$(\lambda x.\,(\lambda x.\,x\,z)\,x)\,x$$

Переменная x — связанная (дважды!) и свободная, а z — свободная.

Свободные и связанные переменные (2)

Определение

Множество $\mathrm{FV}(T)$ ***свободных (free) переменных*** в λ-терме T:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\}; \\ \mathrm{FV}(M\,N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N); \\ \mathrm{FV}(\lambda x.\, M) &= \mathrm{FV}(M) \setminus \{x\}. \end{aligned}$$

Определение

Множество $\mathrm{BV}(T)$ ***связанных (bound) переменных*** в терме T:

$$\begin{aligned} \mathrm{BV}(x) &= \varnothing; \\ \mathrm{BV}(M\,N) &= \mathrm{BV}(M) \cup \mathrm{BV}(N); \\ \mathrm{BV}(\lambda x.\, M) &= \mathrm{BV}(M) \cup \{x\}. \end{aligned}$$

Комбинаторы

Определение

M — ***замкнутый*** λ-***терм*** (или ***комбинатор***), если $FV(M) = \varnothing$. Множество замкнутых λ-термов обозначается Λ^0.

Классические комбинаторы:

$$
\begin{aligned}
\mathbf{I} &= \lambda x.\, x \\
\boldsymbol{\omega} &= \lambda x.\, x\, x \\
\boldsymbol{\Omega} &= \boldsymbol{\omega}\, \boldsymbol{\omega} = (\lambda x.\, x\, x)(\lambda x.\, x\, x) \\
\mathbf{K} &= \lambda x\, y.\, x \\
\mathbf{K}_* &= \lambda x\, y.\, y \\
\mathbf{S} &= \lambda f\, g\, x.\, f\, x\, (g\, x) \\
\mathbf{B} &= \lambda f\, g\, x.\, f\, (g\, x)
\end{aligned}
$$

Функции нескольких переменных, каррирование

- Шонфинкель (1924): функция нескольких переменных может быть описана как конечная последовательность функций одной переменной.
- Пусть $\varphi(x, y)$ — терм, содержащий свободные x и y. Введём, путём последавательных абстракций

$$\Phi_x = \lambda y.\, \varphi(x, y)$$
$$\Phi = \lambda x.\, \Phi_x = \lambda x.\, (\lambda y.\, \varphi(x, y)) = \lambda x\, y.\, \varphi(x, y)$$

- Тогда применение этого Φ к произвольным X и Y может быть выполнена последовательно

$$\Phi\, X\, Y = (\Phi\, X)\, Y = \Phi_X\, Y = (\lambda y.\, \varphi(X, y))\, Y = \varphi(X, Y).$$

- Переход от функции нескольких аргументов к функции, принимающей аргументы «по одному» называется *каррированием*.

Переименование связанных переменных

- Имена *связанных* переменных не важны.
  Переименуем x в y:

$$\lambda \mathrm{x}.\, \mathrm{M}[\mathrm{x}], \quad \lambda \mathrm{y}.\, \mathrm{M}[\mathrm{y}]$$

  Они ведут себя (при подстановках) одинаково:

$$(\lambda \mathrm{x}.\, \mathrm{M}[\mathrm{x}])\, \mathrm{N} = \mathrm{M}[\mathrm{x} := \mathrm{N}], \quad (\lambda \mathrm{y}.\, \mathrm{M}[\mathrm{y}])\, \mathrm{N} = \mathrm{M}[\mathrm{y} := \mathrm{N}]$$

- Поэтому термы часто определяют с точностью до имен связанных переменных. Например,

$$\mathrm{I} = \lambda \mathrm{x}.\, \mathrm{x} \;=\; \lambda \mathrm{y}.\, \mathrm{y}.$$

- Иногда такое переименование называют α-преобразованием и пишут $\mathrm{P} =_\alpha \mathrm{Q}$.

План лекции

Подстановка терма

Определение

$M[x := N]$ обозначает ***подстановку*** N вместо свободных вхождений x в M. Правила подстановки:

$$\begin{aligned}
x[x := N] &= N; \\
y[x := N] &= y; \\
(P\,Q)[x := N] &= (P[x := N])\,(Q[x := N]); \\
(\lambda y.\,P)[x := N] &= \lambda y.\,(P[x := N]),\ y \notin FV(N); \\
(\lambda x.\,P)[x := N] &= (\lambda x.\,P).
\end{aligned}$$

Подразумевается, что $x \not\equiv y$.

Пример

$$((\lambda x.\,(\lambda x.\,x\,z)\,x)\,x)[x := N] = (\lambda x.\,(\lambda x.\,x\,z)\,x)\,N$$

Проблема захвата переменной

Неприятность: $(\lambda y.\, x\, y)[x := y]$ ($y \in \mathrm{FV}(N)$ в четвёртом правиле).

Соглашение Барендрегта

Имена связанных переменных всегда будем выбирать так, чтобы они отличались от имён свободных переменных.

Пример

Вместо

$$y(\lambda x\, y.\, x\, y\, z)$$

будем писать

$$y(\lambda x\, y'.\, x\, y'\, z)$$

Тогда можно использовать подстановку без оговорки о свободных и связанных переменных.

Лемма подстановки

Лемма подстановки

Пусть $M, N, L \in \Lambda$. Предположим $x \not\equiv y$ и $x \notin FV(L)$. Тогда

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Доказательство

Нудная индукция по всем 5 случаям. ■

Преобразования (конверсии): β

Основная схема аксиом для λ-исчисления

Для любых $M, N \in \Lambda$

$$(\lambda x.\, M)N \;=\; M[x := N] \qquad (\beta)$$

- Логические аксиомы и правила:

$$\begin{aligned}
&M \;=\; M;\\
&M = N \;\Rightarrow\; N = M;\\
&M = N, N = L \;\Rightarrow\; M = L;\\
&M = M' \;\Rightarrow\; MZ = M'Z;\\
&M = M' \;\Rightarrow\; ZM = ZM';\\
&M = M' \;\Rightarrow\; \lambda x.\, M = \lambda x.\, M' \quad (\text{правило } \xi).
\end{aligned}$$

- Если $M = N$ доказуемо в λ-исчислении, пишут $\lambda \vdash M = N$.

Преобразования (конверсии): α и η

Иногда вводят:

- схему аксиом α-преобразования:

$$\lambda x.\, M = \lambda y.\, M[x := y] \quad (\alpha)$$

  в предположении, что $y \notin \mathrm{FV}(M)$;

- схему аксиом η-преобразования:

$$\lambda x.\, M\, x = M \qquad (\eta)$$

  в предположении, что $x \notin \mathrm{FV}(M)$.

Преобразования (конверсии): α

Для рассуждений достаточно соглашения Барендрегта, но для компьютерной реализации α-преобразование полезно:

Пример

Пусть $\omega = \lambda x.\, x\, x$ и $1 = \lambda y\, z.\, y\, z$. Тогда

$$\begin{aligned}
\omega\, 1 &= (\lambda x.\, x\, x)(\lambda y\, z.\, y\, z) \\
&= (\lambda y\, z.\, y\, z)(\lambda y\, z.\, y\, z) \\
&= \lambda z.\, (\lambda y\, z.\, y\, z)\, z \\
&= \lambda z.\, (\lambda y\, z'.\, y\, z')\, z \\
&= \lambda z\, z'.\, z\, z' \\
&= \lambda y\, z.\, y\, z \\
&= 1
\end{aligned}$$

Преобразования (конверсии): α

- ***Индексы Де Брауна (De Bruijn)*** представляют альтернативный способ представления термов.
- Связанные переменные не именуются, а индексируются, индекс показывает, сколько лямбд назад переменная была связана:

$$\begin{aligned} \lambda x.\,(\lambda y.\,x\,y) &\leftrightarrow \lambda\,(\lambda\,2\,1) \\ \lambda x.\,x\,(\lambda y.\,x\,y\,y) &\leftrightarrow \lambda\,1\,(\lambda\,2\,1\,1) \end{aligned}$$

- При таком представлении коллизий захвата переменной не возникает.

Преобразования (конверсии): η

- η-преобразование обеспечивает принцип ***экстенсиональности***: две функции считаются экстенсионально эквивалентными, если они дают одинаковый результат при одинаковом вводе:

$$\forall x \ : \ F\,x = G\,x.$$

- Выбирая $y \notin FV(F) \cup FV(G)$, получаем ($\xi$, затем η)

$$\begin{aligned} F\,y &= G\,y \\ \lambda y.\,F\,y &= \lambda y.\,G\,y \\ F &= G \end{aligned}$$

- В Хаскелле так называемый «бесточечный» стиль записи основан на η-преобразовании.

План лекции

Расширения чистого λ-исчисления

- Можно расширить множество λ-термов константами:

$$\Lambda(\mathbb{C}) \quad ::= \quad \mathbb{C} \mid V \mid \Lambda(\mathbb{C})\,\Lambda(\mathbb{C}) \mid \lambda V.\,\Lambda(\mathbb{C})$$

- Например, $\mathbb{C} = \{\mathbf{true}, \mathbf{false}\}$.
- Но нам ещё нужно уметь их использовать. Поэтому лучше

$$\mathbb{C} = \{\mathbf{true}, \mathbf{false}, \mathbf{not}, \mathbf{and}, \mathbf{or}\}$$

- И всё равно, помимо констант нужны дополнительные правила, описывающие работу с ними. Какие?

δ-преобразование: пример

- Всем известные:

$$\begin{aligned}
\mathbf{not\ true} &=_\delta \mathbf{false} \\
\mathbf{not\ false} &=_\delta \mathbf{true}
\end{aligned}$$

$$\begin{aligned}
\mathbf{and\ true\ true} &=_\delta \mathbf{true} \\
\mathbf{and\ true\ false} &=_\delta \mathbf{false} \\
\mathbf{and\ false\ true} &=_\delta \mathbf{false} \\
\mathbf{and\ false\ false} &=_\delta \mathbf{false} \\
&\dots
\end{aligned}$$

- «Внешние» функции над константами порождают новые правила преобразований.

δ-преобразование: обобщение

- Если на множестве термов X (обычно $X \subseteq \mathbb{C}$) задана «внешняя» функция $f : X^k \to \Lambda(\mathbb{C})$, то для неё добавляем δ-правило:
  - выбираем незанятую константу δ_f;
  - для $M_1, \ldots, M_k \in X$ добавляем правило сокращения

$$\delta_f \; M_1 \; \ldots \; M_k \; =_\delta \; f(M_1, \ldots, M_k)$$

- Для одной f — не одно правило, а целая *схема* правил.
- Например, для $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ схемы правил:

$$\begin{aligned}
\mathbf{plus}\ m\ n &=_\delta m + n \\
\mathbf{mult}\ m\ n &=_\delta m \times n \\
\mathbf{equal}\ n\ n &=_\delta \mathbf{true} \\
\mathbf{equal}\ m\ n &=_\delta \mathbf{false},\ \text{если } m \neq n
\end{aligned}$$