Рецепты с рапирой: работаем со связанными переменными без лишних порезов

Николай Кудасов
Семинар МЕТА, ИПС им. А.К. Айламазяна РАН

1 июля 2025

Лаборатория языков программирования и компиляторов

INNOPOLIS
UNIVERSITY

Захват связанных переменных

В присутствии лексически связанных (локальных) переменных, многие действия, начиная с простой подстановки становятся нетривиальными. В λ-исчислении:

$$(\lambda x.\lambda y.x)y =_\beta [x \mapsto y](\lambda y.x) \not\equiv \lambda y.y$$

Это не просто теоретическая проблема! Решение многих практических задач зависит от корректной работы со связанными переменными:

1. проверка и вывод зависимых типов (Friedman и Christiansen 2018)
2. инлайнинг (встраивание функций) и другие гигиенические трансформации в компиляторах, например в GHC (Peyton Jones и Marlow 2002)
3. гигиенические макросы, например в Lean (Ullrich и Moura 2020)
4. формальная метатеория языков программирования (Aydemir и др. 2005)
5. SMT-решатели, решатели теорем, и другие формальные системы

Содержание

Введение в λ-исчисление

Бестиповое λ-исчисление

Нашим рабочим примером будет минималистичный язык — λ-**исчисление**.

1. Это выразительный (полный по Тьюрингу!) чисто-функциональный язык.
2. Позволяет кодировать данные функциями[1].
3. Поощряет рассуждение о корректности программ через переписывание.
4. Используется для представления и поиска доказательств в решателях теорем.
5. Используется как формализм для изучения семантики языков программирования, в частности систем типов (Pierce 2002).

Для нас λ-исчисление служит простым модельным языком, на котором мы можем понять проблемы работы со связанными переменными.

[1] В частности, это может быть использовано для оптимизаций (Voigtländer 2008).

λ-исчисление: синтаксис

Бестиповое λ-исчисление (Pierce 2002, §5.2)

"Программы" λ-исчисления называются **термы** (или λ-**термы**):

1. *переменная* (например, $\boxed{x}$, $\boxed{y}$ или $\boxed{z}$) — это терм
2. *применение* $\boxed{(\mathbf{t_1}\ \mathbf{t_2})}$ терма $\mathbf{t_1}$ к терму $\mathbf{t_2}$ — это терм
3. *абстракция* $\boxed{\lambda x.\mathbf{t}}$ — это терм, если $\mathbf{t}$ — терм (возможно, с переменной x)

λ-исчисление может быть расширено константами, примитивами и пр.

Примеры λ-термов

1. $\lambda x.x$
2. $\lambda x.42$
3. $\lambda x.\lambda y.x + y$
4. $(((\lambda x.\lambda y.y)\ a)\ b)$
5. $f\ (f\ (f\ (f\ (f\ x))))$
6. $\lambda f.(\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x)))$

Бестиповое λ-исчисление (Pierce 2002, §5.2)

"Программы" λ-исчисления называются **термы** (или λ-**термы**):

1. *переменная* (например, $\boxed{x}$, $\boxed{y}$ или $\boxed{z}$) — это терм
2. *применение* $\boxed{(\mathbf{t_1}\ \mathbf{t_2})}$ терма $\mathbf{t_1}$ к терму $\mathbf{t_2}$ — это терм
3. *абстракция* $\boxed{\lambda x.\mathbf{t}}$ — это терм, если $\mathbf{t}$ — терм (возможно, с переменной x)

λ-исчисление может быть расширено константами, примитивами и пр.

Примеры λ-термов

1. $\lambda x.x$
2. $\lambda x.42$
3. $\lambda x.\lambda y.x + y$
4. $(((\lambda x.\lambda y.y)\ a)\ b)$
5. $f\ (f\ (f\ (f\ (f\ x))))$
6. $\lambda f.(\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x)))$

λ-исчисление: синтакстические соглашения

Следующие соглашения распространены[2]:

1. Применение лево-ассоциативно:
   - $\boxed{s\ t\ u}$ — это то же, что $\boxed{(s\ t)\ u}$
   - $\boxed{f\ (g\ x\ y)\ z}$ — это то же, что $\boxed{(f\ ((g\ x)\ y))\ z}$
2. λx связывает переменную x до конца строки (либо до закрывающейся скобки):
   - $\boxed{\lambda x.\lambda y.x\ y\ x}$ — это то же, что $\boxed{\lambda x.(\lambda y.(x\ y)\ x)}$
   - $\boxed{\lambda x.(\lambda y.x)\ y\ x}$ — это то же, что $\boxed{\lambda x.(((\lambda y.x)\ y)\ x)}$

[2] Но существуют и другие соглашения, например нотация Кривина.

λ-исчисление: синтакстические соглашения

Следующие соглашения распространены[2]:

1. Применение лево-ассоциативно:
   - $\boxed{s\ t\ u}$ — это то же, что $\boxed{(s\ t)\ u}$
   - $\boxed{f\ (g\ x\ y)\ z}$ — это то же, что $\boxed{(f\ ((g\ x)\ y))\ z}$
2. λx связывает переменную x до конца строки (либо до закрывающейся скобки):
   - $\boxed{\lambda x.\lambda y.x\ y\ x}$ — это то же, что $\boxed{\lambda x.(\lambda y.(x\ y)\ x)}$
   - $\boxed{\lambda x.(\lambda y.x)\ y\ x}$ — это то же, что $\boxed{\lambda x.(((\lambda y.x)\ y)\ x)}$

[2] Но существуют и другие соглашения, например нотация Кривина.

λ-исчисление: синтакстические соглашения

Следующие соглашения распространены[2]:

1. Применение лево-ассоциативно:
   - $\boxed{s\ t\ u}$ — это то же, что $\boxed{(s\ t)\ u}$
   - $\boxed{f\ (g\ x\ y)\ z}$ — это то же, что $\boxed{(f\ ((g\ x)\ y))\ z}$
2. λx связывает переменную x до конца строки (либо до закрывающейся скобки):
   - $\boxed{\lambda x.\lambda y.x\ y\ x}$ — это то же, что $\boxed{\lambda x.(\lambda y.(x\ y)\ x)}$
   - $\boxed{\lambda x.(\lambda y.x)\ y\ x}$ — это то же, что $\boxed{\lambda x.(((\lambda y.x)\ y)\ x)}$

[2] Но существуют и другие соглашения, например нотация Кривина.

λ-исчисление: синтакстические соглашения

Следующие соглашения распространены[2]:

1. Применение лево-ассоциативно:
   - $\boxed{s\ t\ u}$ — это то же, что $\boxed{(s\ t)\ u}$
   - $\boxed{f\ (g\ x\ y)\ z}$ — это то же, что $\boxed{(f\ ((g\ x)\ y))\ z}$
2. λx связывает переменную x до конца строки (либо до закрывающейся скобки):
   - $\boxed{\lambda x.\lambda y.x\ y\ x}$ — это то же, что $\boxed{\lambda x.(\lambda y.(x\ y)\ x)}$
   - $\boxed{\lambda x.(\lambda y.x)\ y\ x}$ — это то же, что $\boxed{\lambda x.(((\lambda y.x)\ y)\ x)}$

[2]Но существуют и другие соглашения, например нотация Кривина.

λ-исчисление: области видимости переменных

В λ-терме каждое вхождение переменной либо **свободно**, либо **связано**.

1. Вхождение переменной z **связано** если оно входит в абстракцию λz.
2. Иначе вхождение переменной z **свободно**.

λ-терм называется **замкнутым**, если в нём нет свободных переменных.

Примеры свободных и связанных переменных

Свободные переменные — **чёрные**, а связанные переменные и абстракции индексированы:

1. $\lambda x_1.x_1$
2. $\lambda x_1.\mathbf{z}$
3. $\lambda x_1.\lambda y_2.x_1 + y_2$
4. $(((\lambda x_1.\lambda x_2.x_2)\ \mathbf{a})\ \mathbf{b})$
5. $\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,\mathbf{x}))))$
6. $\lambda f_1.(\lambda x_2.(f_1(x_2\ x_2)))\ (\lambda x_3.(f_1(x_3\ x_3)))$

Какие из вышеперечисленных термов замкнуты?

λ-исчисление: области видимости переменных

В λ-терме каждое вхождение переменной либо **свободно**, либо **связано**.

1. Вхождение переменной z **связано** если оно входит в абстракцию λz.
2. Иначе вхождение переменной z **свободно**.

λ-терм называется **замкнутым**, если в нём нет свободных переменных.

Примеры свободных и связанных переменных

Свободные переменные — **чёрные**, а связанные переменные и абстракции индексированы:

1. $\lambda x_1.x_1$
2. $\lambda x_1.\mathbf{z}$
3. $\lambda x_1.\lambda y_2.x_1 + y_2$
4. $(((\lambda x_1.\lambda x_2.x_2)\ \mathbf{a})\ \mathbf{b})$
5. $\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,\mathbf{x}))))$
6. $\lambda f_1.(\lambda x_2.(f_1(x_2\ x_2)))\ (\lambda x_3.(f_1(x_3\ x_3)))$

Какие из вышеперечисленных термов замкнуты?

λ-исчисление: области видимости переменных

В λ-терме каждое вхождение переменной либо **свободно**, либо **связано**.

1. Вхождение переменной z **связано** если оно входит в абстракцию λz.
2. Иначе вхождение переменной z **свободно**.

λ-терм называется **замкнутым**, если в нём нет свободных переменных.

Примеры свободных и связанных переменных

Свободные переменные — **чёрные**, а связанные переменные и абстракции индексированы:

1. $\lambda x_1.x_1$
2. $\lambda x_1.\mathbf{z}$
3. $\lambda x_1.\lambda y_2.x_1 + y_2$
4. $(((\lambda x_1.\lambda x_2.x_2)\ \mathbf{a})\ \mathbf{b})$
5. $\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,(\mathbf{f}\,\mathbf{x}))))$
6. $\lambda f_1.(\lambda x_2.(f_1(x_2\ x_2)))\ (\lambda x_3.(f_1(x_3\ x_3)))$

Какие из вышеперечисленных термов замкнуты?

Вычисление λ-термов определяется взаимодействием абстракции и применения:

$$(\lambda \mathbf{x}.\mathbf{t})\ \mathbf{u} \quad\longrightarrow\quad [\mathbf{x} \mapsto \mathbf{u}]\mathbf{t}$$

Это правило называется β-редукцией:

- $\mathbf{x}$ — это *метапеременная*, значением которой может выступать переменная
- $\mathbf{t}$ и $\mathbf{u}$ — *метапеременная*, значениями которых могут выступать λ-термы
- $[\mathbf{x} \mapsto \mathbf{u}]\mathbf{t}$ означает “заменить все вхождения $\mathbf{x}$ на $\mathbf{u}$ в терме $\mathbf{t}$”

Примеры β-редукции

$$\begin{array}{rcll}
(\lambda x.x)\ y & \longrightarrow & [x \mapsto y]x & \equiv\ y \\
(\lambda y.\lambda x.y)\ (\lambda z.z) & \longrightarrow & [y \mapsto (\lambda z.z)](\lambda x.y) & \equiv\ \lambda x.\lambda z.z \\
(\lambda z.z\ (\lambda z.z))(u\ r) & \longrightarrow & [z \mapsto u\ r](z\ (\lambda z.z)) & \equiv\ u\ r\ (\lambda z.z)
\end{array}$$

Вычисление λ-термов определяется взаимодействием абстракции и применения:

$$(\lambda \mathbf{x}.\mathbf{t})\ \mathbf{u} \quad \longrightarrow \quad [\mathbf{x} \mapsto \mathbf{u}]\mathbf{t}$$

Это правило называется β-редукцией:

- $\mathbf{x}$ — это *метапеременная*, значением которой может выступать переменная
- $\mathbf{t}$ и $\mathbf{u}$ — *метапеременная*, значениями которых могут выступать λ-термы
- $[\mathbf{x} \mapsto \mathbf{u}]\mathbf{t}$ означает "заменить все вхождения $\mathbf{x}$ на $\mathbf{u}$ в терме $\mathbf{t}$"

Примеры β-редукции

$$\begin{array}{rcll}
(\lambda x.x)\ y & \longrightarrow & [x \mapsto y]x & \equiv \quad y \\
(\lambda y.\lambda x.y)\ (\lambda z.z) & \longrightarrow & [y \mapsto (\lambda z.z)](\lambda x.y) & \equiv \quad \lambda x.\lambda z.z \\
(\lambda z.z\ (\lambda z.z))(u\ r) & \longrightarrow & [z \mapsto u\ r](z\ (\lambda z.z)) & \equiv \quad u\ r\ (\lambda z.z)
\end{array}$$

λ-исчисление: подстановка (попытка 1)

Следующее определение подстановки **неверно**. Почему?

$$
\begin{aligned}
[\mathbf{x} \mapsto \mathbf{s}]\mathbf{x} &= \mathbf{s} \\
[\mathbf{x} \mapsto \mathbf{s}]\mathbf{y} &= \mathbf{y} \quad \text{if } \mathbf{x} \neq \mathbf{y} \\
[\mathbf{x} \mapsto \mathbf{s}](\lambda\mathbf{y}.\mathbf{t_1}) &= \lambda\mathbf{y}.[\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1} \\
[\mathbf{x} \mapsto \mathbf{s}](\mathbf{t_1}\ \mathbf{t_2}) &= [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}\ [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}
\end{aligned}
$$

λ-исчисление: подстановка (попытка 1)

Следующее определение подстановки **неверно**. Почему?

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{x} = \mathbf{s}$$

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{y} = \mathbf{y} \quad \text{if } \mathbf{x} \neq \mathbf{y}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\lambda \mathbf{y}.\mathbf{t_1}) = \lambda \mathbf{y}.[\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\mathbf{t_1}\ \mathbf{t_2}) = [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}\ [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}$$

Подсказка: каков результат $[x \mapsto f\ y](\lambda x.x)$?

λ-исчисление: подстановка (попытка 2)

Следующее определение подстановки **неверно**. Почему?

$$
\begin{aligned}
[\mathbf{x} \mapsto \mathbf{s}]\mathbf{x} &= \mathbf{s} \\
[\mathbf{x} \mapsto \mathbf{s}]\mathbf{y} &= \mathbf{y} \quad \text{if } \mathbf{x} \neq \mathbf{y} \\
[\mathbf{x} \mapsto \mathbf{s}](\lambda \mathbf{y}.\mathbf{t_1}) &= \begin{cases} \lambda \mathbf{y}.\mathbf{t_1}, & \text{если } \mathbf{x} = \mathbf{y} \\ \lambda \mathbf{y}.[\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}, & \text{иначе} \end{cases} \\
[\mathbf{x} \mapsto \mathbf{s}](\mathbf{t_1}\ \mathbf{t_2}) &= [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}\ [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_2}
\end{aligned}
$$

λ-исчисление: подстановка (попытка 2)

Следующее определение подстановки **неверно**. Почему?

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{x} = \mathbf{s}$$

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{y} = \mathbf{y} \quad \text{if } \mathbf{x} \neq \mathbf{y}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\lambda \mathbf{y}.\mathbf{t_1}) = \begin{cases} \lambda \mathbf{y}.\mathbf{t_1}, & \text{если } \mathbf{x} = \mathbf{y} \\ \lambda \mathbf{y}.[\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}, & \text{иначе} \end{cases}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\mathbf{t_1}\ \mathbf{t_2}) = [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}\ [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_2}$$

Каков результат $[x \mapsto f\ y](\lambda y.x)$?

λ-исчисление: подстановка (попытка 3)

Следующее определение подстановки **неполно**. Почему?

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{x} = \mathbf{s}$$

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{y} = \mathbf{y} \quad \text{если } \mathbf{x} \neq \mathbf{y}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\lambda \mathbf{y}.\mathbf{t_1}) = \begin{cases} \lambda \mathbf{y}.\mathbf{t_1}, & \text{если } \mathbf{x} = \mathbf{y} \\ \lambda \mathbf{y}.[\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}, & \text{если } \mathbf{x} \neq \mathbf{y} \text{ и } \mathbf{y} \text{ не входит свободно в } \mathbf{s} \end{cases}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\mathbf{t_1}\ \mathbf{t_2}) = [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}\ [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_2}$$

λ-исчисление: подстановка (попытка 3)

Следующее определение подстановки **неполно**. Почему?

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{x} = \mathbf{s}$$

$$[\mathbf{x} \mapsto \mathbf{s}]\mathbf{y} = \mathbf{y} \quad \text{если } \mathbf{x} \neq \mathbf{y}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\lambda \mathbf{y}.\mathbf{t_1}) = \begin{cases} \lambda \mathbf{y}.\mathbf{t_1}, & \text{если } \mathbf{x} = \mathbf{y} \\ \lambda \mathbf{y}.[\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}, & \text{если } \mathbf{x} \neq \mathbf{y} \text{ и } \mathbf{y} \text{ не входит свободно в } \mathbf{s} \end{cases}$$

$$[\mathbf{x} \mapsto \mathbf{s}](\mathbf{t_1}\ \mathbf{t_2}) = [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_1}\ [\mathbf{x} \mapsto \mathbf{s}]\mathbf{t_2}$$

Каков результат $[x \mapsto f\ y](\lambda y.x)$?

λ-исчисление: α-эквивалентность

Имена связанных переменных не важны!

Термы $\mathbf{t_1}$ и $\mathbf{t_2}$ α-**эквивалентны**, если $\mathbf{t_2}$ можно получить из $\mathbf{t_1}$ (и наоборот) переименованием 0 или более связанных переменных.

Примеры α-эквивалентных и не эквивалетных термов

1. $(\lambda x.x)\ (\lambda x.x)$ $\quad$ $(\lambda y.y)\ (\lambda z.z)$
2. $\lambda z.\lambda x.\lambda y.x\ (y\ z)$ $\quad$ $\lambda a.\lambda b.\lambda c.b\ (c\ a)$
3. $(\lambda x.y)\ (\lambda y.y)$ $\quad$ $(\lambda z.y)\ (\lambda x.x)$
4. $\lambda x.y$ $\quad$ $\lambda x.z$
5. $(\lambda x.y)\ (\lambda y.y)$ $\quad$ $(\lambda y.y)\ (\lambda x.x)$

λ-исчисление: α-эквивалентность

Имена связанных переменных не важны!

Термы $\mathbf{t_1}$ и $\mathbf{t_2}$ α-**эквивалентны**, если $\mathbf{t_2}$ можно получить из $\mathbf{t_1}$ (и наоборот) переименованием 0 или более связанных переменных.

Примеры α-эквивалентных и не эквивалетных термов

1. $(\lambda x.x)\ (\lambda x.x)$ α-эквивалентен $(\lambda y.y)\ (\lambda z.z)$
2. $\lambda z.\lambda x.\lambda y.x\ (y\ z)$ $\lambda a.\lambda b.\lambda c.b\ (c\ a)$
3. $(\lambda x.y)\ (\lambda y.y)$ $(\lambda z.y)\ (\lambda x.x)$
4. $\lambda x.y$ $\lambda x.z$
5. $(\lambda x.y)\ (\lambda y.y)$ $(\lambda y.y)\ (\lambda x.x)$

λ-исчисление: α-эквивалентность

Имена связанных переменных не важны!

Термы $\mathbf{t_1}$ и $\mathbf{t_2}$ α-**эквивалентны**, если $\mathbf{t_2}$ можно получить из $\mathbf{t_1}$ (и наоборот) переименованием 0 или более связанных переменных.

Примеры α-эквивалентных и не эквивалетных термов

1. $(\lambda x.x)\ (\lambda x.x)$ α-эквивалентен $(\lambda y.y)\ (\lambda z.z)$
2. $\lambda z.\lambda x.\lambda y.x\ (y\ z)$ α-эквивалентен $\lambda a.\lambda b.\lambda c.b\ (c\ a)$
3. $(\lambda x.y)\ (\lambda y.y)$ $(\lambda z.y)\ (\lambda x.x)$
4. $\lambda x.y$ $\lambda x.z$
5. $(\lambda x.y)\ (\lambda y.y)$ $(\lambda y.y)\ (\lambda x.x)$

λ-исчисление: α-эквивалентность

Имена связанных переменных не важны!

Термы $\mathbf{t_1}$ и $\mathbf{t_2}$ α-**эквивалентны**, если $\mathbf{t_2}$ можно получить из $\mathbf{t_1}$ (и наоборот) переименованием 0 или более связанных переменных.

Примеры α-эквивалентных и не эквивалетных термов

1. $(\lambda x.x)\ (\lambda x.x)$ α-эквивалентен $(\lambda y.y)\ (\lambda z.z)$
2. $\lambda z.\lambda x.\lambda y.x\ (y\ z)$ α-эквивалентен $\lambda a.\lambda b.\lambda c.b\ (c\ a)$
3. $(\lambda x.y)\ (\lambda y.y)$ α-эквивалентен $(\lambda z.y)\ (\lambda x.x)$
4. $\lambda x.y$ $\lambda x.z$
5. $(\lambda x.y)\ (\lambda y.y)$ $(\lambda y.y)\ (\lambda x.x)$

λ-исчисление: α-эквивалентность

Имена связанных переменных не важны!

Термы $\mathbf{t_1}$ и $\mathbf{t_2}$ α-**эквивалентны**, если $\mathbf{t_2}$ можно получить из $\mathbf{t_1}$ (и наоборот) переименованием 0 или более связанных переменных.

Примеры α-эквивалентных и не эквивалетных термов

1. $(\lambda x.x)\,(\lambda x.x)$ α-эквивалентен $(\lambda y.y)\,(\lambda z.z)$
2. $\lambda z.\lambda x.\lambda y.x\,(y\;z)$ α-эквивалентен $\lambda a.\lambda b.\lambda c.b\,(c\;a)$
3. $(\lambda x.y)\,(\lambda y.y)$ α-эквивалентен $(\lambda z.y)\,(\lambda x.x)$
4. $\lambda x.y$ **не** α-эквивалентен $\lambda x.z$
5. $(\lambda x.y)\,(\lambda y.y)$ $(\lambda y.y)\,(\lambda x.x)$

Имена связанных переменных не важны!

Термы $\mathbf{t_1}$ и $\mathbf{t_2}$ α-**эквивалентны**, если $\mathbf{t_2}$ можно получить из $\mathbf{t_1}$ (и наоборот) переименованием 0 или более связанных переменных.

Примеры α-эквивалентных и не эквивалетных термов

1. $(\lambda x.x)\ (\lambda x.x)$ α-эквивалентен $(\lambda y.y)\ (\lambda z.z)$
2. $\lambda z.\lambda x.\lambda y.x\ (y\ z)$ α-эквивалентен $\lambda a.\lambda b.\lambda c.b\ (c\ a)$
3. $(\lambda x.y)\ (\lambda y.y)$ α-эквивалентен $(\lambda z.y)\ (\lambda x.x)$
4. $\lambda x.y$ **не** α-эквивалентен $\lambda x.z$
5. $(\lambda x.y)\ (\lambda y.y)$ **не** α-эквивалентен $(\lambda y.y)\ (\lambda x.x)$

λ-исчисление и Python (пример 1)

Сравните β-редукцию λ-термов

$$(\lambda x.\lambda y.x)\ 1\ 2 \quad \longrightarrow \quad (\lambda y.1)\ 2 \quad \longrightarrow \quad 1$$

С встраиванием функций (рефакторингом) в программе на Python:

```python
def f(x):
  def g(y):
    return x
  return g

print(f(1)(2))
```

```python
# после встраивания f

def g1(y):
  return 1

print(g1(2))
```

```python
# после встраивания f
# а затем — g1



print(1)
```

λ-исчисление и Python (пример 2)

Сравните β-редукцию λ-термов со **свободными переменными**

$$(\lambda x.\lambda y.x)\ y\ x \quad \longrightarrow \quad (\lambda z.y)\ x \quad \longrightarrow \quad y$$

С встраиванием функций (рефакторингом) в программе на Python с **глобальными/импортированными переменными**:

```python
def f(x):
  def g(y):
    return x
  return g

print(f(y)(x))
```

```python
# после встраивания f

def g1(z): # y → z (!)
  return y

print(g1(x))
```

```python
# после встраивания f
# а затем — g1



print(y)
```

Наивная реализация

Абстрактный синтаксис в Haskell (арифметические выражения)

Часто выражения задают при помощи алгебраических типов:

```
data Expr
  = Lit Int          -- 0, 1, 2, ...
  | Add Expr Expr    -- e₁ + e₂
  | Mul Expr Expr    -- e₁ × e₂
```

С таким представлением достаточно легко работать:

```
example :: Expr
example = Mul (Add (Lit 2) (Lit 3)) Lit 5     -- (2 + 3) × 5

eval :: Expr Int
eval (Lit n) = n
eval (Add l r) = eval l + eval r
eval (Mul l r) = eval l * eval r
```

Абстрактный синтаксис в Haskell (λ-исчисление)

Аналогично, мы можем представить термы λ-исчисления. Часто такие типы термов параметризуют типом идентификаторов переменных:

```
data Term var
  = Var var                          -- x
  | Lam var (Term var)               -- λx.t
  | App (Term var) (Term var)        -- (t₁ t₂)
```

Строить термы достаточно просто:

```
-- two := λs.λz.s (s z)
two :: Term String
two = Lam "s" (Lam "z" (App (Var "s") (App (Var "s") (Var "z"))))
```

Но насколько сложно реализовать подстановку?

Абстрактный синтаксис в Haskell (λ-исчисление)

Аналогично, мы можем представить термы λ-исчисления. Часто такие типы термов параметризуют типом идентификаторов переменных:

```
data Term var
  = Var var                        -- x
  | Lam var (Term var)             -- λx.t
  | App (Term var) (Term var)      -- (t₁ t₂)
```

Строить термы достаточно просто:

```
-- two := λs.λz.s (s z)
two :: Term String
two = Lam "s" (Lam "z" (App (Var "s") (App (Var "s") (Var "z"))))
```

Но насколько сложно реализовать подстановку?

Подстановка (базовая, неверная)

Подстановка реализуется в целом несложно, но довольно легко совершить ошибку и допустить захват связанных переменных!

```
substitute :: Eq var => (var, Term var) -> Term var -> Term var
substitute (x, u) t = go t
  where
    go (Var y)
      | x == y     = u                       -- [x ↦ u]x = u
      | otherwise  = t                       -- [x ↦ u]y = t
    go (App t1 t2) = App (go t1) (go t2) -- [x ↦ u](t₁ t₂) = [x ↦ u]t₁ [x ↦ u]t₂
    go (Lam z body)
      | x == z     = t                       -- [x ↦ u](λx.t) = λx.t
      | otherwise  = Lam z (go body)         -- [x ↦ u](λz.t) = λz.[x ↦ u]t
```

Код выше содержит ошибку. Можете ли вы её найти?

Проблемы наивной подстановки

Наивная подстановка[3] имеет ряд проблем:

1. Переименование переменных требует глобального потока имён. (нельзя просто распараллелить на большой программе)
2. Далеко не все переменные необходимо переименовывать. (неэффективна)
3. Разработчику легко можно забыть переименовать переменную. (не типо-безопасна)
4. Реализация нужна для каждой новой вариации/части языка. (не переиспользуема/универсальна)

[3] См. “кувалду” (Peyton Jones и Marlow 2002, §4.1).

Представления связанных переменных

Неполный список представлений связанных переменных:

	Безопасность	Производ.	Универс.
Наивная подстановка	**НЕТ**	**НЕТ**	**НЕТ**
de Bruijn 1972	**НЕТ**	**ИНОГДА**	**НЕТ**
Bird и Paterson 1999	**ДА**	**НЕТ**	**НЕТ**
“Рапира” (Peyton Jones и Marlow 2002)	**НЕТ**	**ДА**	**НЕТ**
PHOAS (Chlipala 2008)	**ДА**	**ДА**	**НЕТ**[4]
“Foil” (Maclaurin, Radul и Paszke 2023)	**ДА**	**ДА**	**НЕТ**
“Free scoped monads” (Kudasov 2025)	**ДА**	**НЕТ**	**ДА**
“Free Foil” (Kudasov и др. 2024)	**ДА**	**ДА**[5]	**ДА**

[4] Kmett 2008: http://comonad.com/reader/2008/rotten-bananas/

[5] Но есть замедление по сравнению с Foil.

Рапира

Рапира (основная идея)

“Рапира” (Peyton Jones и Marlow 2002, §4.2) предлагает существенное улучшение:

1. Отслеживает множество переменных в текущей области видимости (*скоуп*).
2. Генерирует *локально* свежое имя (а не глобальное).
3. Скоупы реализованы эффективно через персистентные структуры данных.
4. Термы поддерживают “конвенцию Барендрегта” (нет затемнения имён).

Рапира (основная идея)

“Рапира” (Peyton Jones и Marlow 2002, §4.2) предлагает существенное улучшение:

1. Отслеживает множество переменных в текущей области видимости (*скоуп*).
2. Генерирует *локально* свежое имя (а не глобальное).
3. Скоупы реализованы эффективно через персистентные структуры данных.
4. Термы поддерживают “конвенцию Барендрегта” (нет затемнения имён).

И тем не менее, есть проблемы:

> Names turn out to be one of the Hard Things in writing compilers as well. In the Dex compiler, for instance, we’ve been following GHC’s version of the Barendregt convention, “the rapier” [9]. It‘s elegant and it’s fast. It’s also stateless, which is crucial for caching and concurrency.
>
> But it‘s *really* easy to screw up. We’ve messed it up again[1] and again[2] and again[3] and again[4] and again[5] and again.[6] This had become one of the biggest barriers to implementing new language ideas and onboarding new people. Worse, it made us hesitate to use name-based indirection in places it would have been helpful.

Авторы языка Dex о “рапире”. (Maclaurin, Radul и Paszke 2023)

Рапира без порезов (Foil)

Foil: типы и области видимости

“Foil” (Maclaurin, Radul и Paszke 2023) накладывает безопасный интерфейс на рапиру, избавляя пользователя от порезов в рантайме[6]:

1. `Name n` — тип индентификатора в скоупе n
2. `NameBinder n l` — тип идентификатора, расширяющего скоуп n до скоупа l
3. `Scope n` — множество идентификаторов в скоупе n
4. `DExt n l` — ограничение, гарантирующее, что l расширяет n

Для безопасной работы с областями видимости, “Foil” полагается на параметрический полиморфизм ранга 2:

```
withRefreshed :: Scope o
              -> Name i
              -> (forall o'. DExt o o' => NameBinder o o' -> r)
              -> r
```

[6] Но по моему опыту, этот подход привносит свои неудобства на этап компиляции.

Foil: переименуемость (пользовательский код)

"Foil" можно использовать, чтобы определить синтаксис, статически аннотированный областями видимости:

```
data Expr n where
  VarE :: Name n -> Expr n                         -- variable: x
  AppE :: Expr n -> Expr n -> Expr n               -- application: (e1 e2)
  LamE :: NameBinder n l -> Expr l -> Expr n       -- abstraction: λx.e
```

Следующий код **доказывает** компилятору свойство, благодаря которому можно безопасно "погружать" выражения в расширенный контекст:

```
instance Sinkable Expr where
  -- sinkabilityProof :: (Name n -> Name l) -> Expr n -> Expr l
  sinkabilityProof rename (VarE v) = VarE (rename v)
  sinkabilityProof rename (AppE f e) =
    AppE (sinkabilityProof rename f) (sinkabilityProof rename e)
  sinkabilityProof rename (LamE binder body) =
    extendRenaming rename binder $ \rename' binder' ->
      LamE binder' (sinkabilityProof rename' body)
```

Foil: подстановка (пользовательский код)

За счёт подвинутого использования системы типов, “забыть” переименовать переменные невозможно:

```
substitute :: Distinct o => Scope o -> Substitution Expr i o -> Expr i -> Expr o
substitute scope subst = \case
  VarE name -> lookupSubst subst name
  AppE f x -> AppE (substitute scope subst f) (substitute scope subst x)
  LamE binder body -> withRefreshed scope binder $ \extendSubst binder' ->
    let subst' = extendSubst subst
        scope' = extendScope binder' scope
        body' = substitute scope' subst' body
      in LamE binder' body'
```

Foil: подстановка (пользовательский код)

За счёт подвинутого использования системы типов, “забыть” переименовать переменные невозможно:

```
substitute :: Distinct o => Scope o -> Substitution Expr i o -> Expr i -> Expr o
substitute scope subst = \case
  VarE name -> lookupSubst subst name
  AppE f x -> AppE (substitute scope subst f) (substitute scope subst x)
  LamE binder body -> withRefreshed scope binder $ \extendSubst binder' ->
    let subst' = extendSubst subst
        scope' = extendScope binder' scope
        body' = substitute scope' subst' body
      in LamE binder' body'
```

Но пользователь всё ещё вынужден этот код писать (и бороться с тайпчекером) для каждой вариации языка. И это мы ещё не говорим о проверке α-эквивалентности...

Ограничения подхода Foil

“Foil” безопасен, но требует многого от разработчика:

1. реализацию подстановки
2. реализацию α-эквивалентности (причём это удивительно нетривиально!)
3. объявление вариации абстракного синтаксиса (с учётом областей видимости)
4. функции конвертации из/в новое представление
5. автоматическое доказательство для “погружения” термов

Также изначально “Foil” не предлагает механизмов работы со сложными образцами.

Рапира в меню (Free Foil)

Free Foil: обобщение рапиры без порезов

Free Foil (Kudasov 2024) объединяет “Foil” и “data types à la carte”[7]:

```
data ScopedAST sig n where
  ScopedAST :: NameBinder n l -> AST sig l -> ScopedAST sig n

data AST sig n where
  Var  :: Name n -> AST sig n
  Node :: sig (ScopedAST sig n) (AST sig n) -> AST sig n
```

Здесь `AST sig n` — это обобщённый тип термов свободно сгенерированный из сигнатуры `sig`. Например, у λ-исчисления сигнатура выглядит так:

```
data ExprSig scope term = App term term | Lam scope
```

Безопасный абстрактный синтаксис для λ-исчисления получается так:

```
type Expr n = AST ExprSig n
```

[7]См. (Swierstra 2008)

Free Foil

“Free Foil” позволяет написать ряд алгоритмов один раз и переиспользовать их для любого целевого языка:

1. подстановка без захвата имён
2. проверка α-эквивалентности
3. типы безопасного абстракного синтаксиса
4. алгоритмы проверки типов[8]
5. алгоритмы унификации высшего порядка[9]

[8] https://github.com/evermake/free-foil-typecheck
[9] https://github.com/fedor-ivn/free-foil-hou

Обсуждение

Итого

- Работа с именами — это сложно.
- При этом многим инструментам это нужно.
- Рапира предлагает элегантный и эффективный подход (используется в GHC, Agda).
- Foil предлагает безопасный интерфейс вокруг рапиры (используется в Dex).
- Free Foil предлагает реализовывать переиспользуемые алгоритмы с именами.

Спасибо за внимание!

Список литературы i

- Aydemir, Brian E. и др. (2005). **«Mechanized Metatheory for the Masses: The PoplMark Challenge».** В: *Theorem Proving in Higher Order Logics*. Под ред. Joe Hurd и Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, с. 50—65. ISBN: 978-3-540-31820-0.
- Bird, Richard S. и Ross Paterson (1999). **«de Bruijn notation as a nested datatype».** В: *Journal of Functional Programming* 9.1, с. 77—91. DOI: 10.1017/S0956796899003366.
- Chlipala, Adam (сент. 2008). **«Parametric Higher-Order Abstract Syntax for Mechanized Semantics».** В: *SIGPLAN Not.* 43.9, с. 143—156. ISSN: 0362-1340. DOI: 10.1145/1411203.1411226.

- de Bruijn, N.G (1972). **«Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem».** В: *Indagationes Mathematicae (Proceedings)* 75.5, с. 381—392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.
- Friedman, Daniel P и David Thrane Christiansen (2018). ***The Little Typer.*** MIT Press.
- Kmett, Edward (март 2008). ***Rotten Bananas.*** Accessed: 2023-01-21. URL: http://comonad.com/reader/2008/rotten-bananas/.

- Kudasov, Nikolai (2024). ***Free Monads, Intrinsic Scoping, and Higher-Order Preunification.*** To appear in *TFP 2024*. Orange, NJ, USA. arXiv: 2204.05653 [cs.LO].
- — (2025). **«Free Monads, Intrinsic Scoping, and Higher-Order Preunification».** В: *Trends in Functional Programming*. Под ред. Jason Hemann и Stephen Chang. Cham: Springer Nature Switzerland, с. 22—54. ISBN: 978-3-031-74558-4.
- Kudasov, Nikolai и др. (2024). **«Free Foil: Generating Efficient and Scope-Safe Abstract Syntax».** В: *2024 4th International Conference on Code Quality (ICCQ)*, с. 1—16. DOI: 10.1109/ICCQ60895.2024.10576867.

Список литературы iv

- Maclaurin, Dougal, Alexey Radul и Adam Paszke (2023). **«The Foil: Capture-Avoiding Substitution With No Sharp Edges».** B: *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*. IFL '22. Copenhagen, Denmark: Association for Computing Machinery. ISBN: 9781450398312. DOI: 10.1145/3587216.3587224. URL: https://doi.org/10.1145/3587216.3587224.
- Peyton Jones, Simon и Simon Marlow (июль 2002). **«Secrets of the Glasgow Haskell Compiler inliner».** B: *Journal of Functional Programming* 12, c. 393—434. URL: https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/.

Список литературы v

- Pierce, B.C. (2002). ***Types and Programming Languages.*** The MIT Press. MIT Press. ISBN: 9780262162098. URL: https://books.google.ru/books?id=ULT4DwAAQBAJ.
- Swierstra, Wouter (2008). **«Data types à la carte».** В: *Journal of Functional Programming* 18.4, с. 423—436. DOI: 10.1017/S0956796808006758.
- Ullrich, Sebastian и Leonardo de Moura (2020). **«Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages».** В: *Automated Reasoning.* Под ред. Nicolas Peltier и Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, с. 167—182. ISBN: 978-3-030-51054-1.

Voigtländer, Janis (2008). **«Asymptotic Improvement of Computations over Free Monads».** В: *Mathematics of Program Construction*. Под ред. Philippe Audebaud и Christine Paulin-Mohring. Berlin, Heidelberg: Springer Berlin Heidelberg, с. 388—403. ISBN: 978-3-540-70594-9. DOI: 10.1007/s10817-011-9225-2.