Теория ФП

Лекция 3.

ВМК МГУ, 2010. Контакты: ximaera@cs.msu.su, http://haskell.secsem.ru

Комбинаторная логика

- Хаскелл Карри, Мозес Шонфинкель
  - 40-е гг. XX в.
  - Теория вычисления без переменных
  - Упрощение λ-исчисления
    - Казалось бы, что там ещё упрощать?

База λ-исчисления (1/2)

- Множество термов Λ строится по 4 правилам
  - Два правила определяют алфавит
    - $I \subset \Lambda$ – переменные
    - $L \in \Lambda \Rightarrow (L) \in \Lambda$ – скобки
  - Другие два определяют операции, допустимые при моделировании
    - $i \in I,\ L \in \Lambda \Rightarrow \lambda i.L \in \Lambda$
    - $L, K \in \Lambda \Rightarrow (LK) \in \Lambda$

База λ-исчисления (2/2)

- Операции, допустимые при моделировании с использованием λ-исчисления:
  - $L, K \in \Lambda \Rightarrow (LK) \in \Lambda$ – аппликация: подстановка выражения K в L; применение, от англ. «application», функции L к выражению K
  - $x \in I, L \in \Lambda \Rightarrow \lambda x.L \in \Lambda$ – абстракция: замена глобальной (свободной) переменной x на локальную (связанную), которую можно затем заменять на любое абстрактное выражение

Упрощение λ-исчисления

- Хаскелл Карри предложил избавиться от абстракции

  L, K∈Λ => (LK)∈Λ – аппликация: подстановка выражения K в L; единственная операция в комбинаторной логике

- Комбинаторные термы строятся не на основе абстрагирования переменных, а на основе базиса предопределенных функций (базовых комбинаторов) и комбинирования функций друг с другом

Базовый комбинатор (1/2)

- Базовый комбинатор – постоянный, заранее определённый объект, не содержащий свободных переменных, декларирующий правила комбинирования объектов друг с другом, представленный как правило:

  $$\mathrm{P}\ x_1 x_2 x_3 \ldots x_N = \mathrm{E}$$

  где выражение E составлено только из переменных $x_1 \ldots x_N$ – аргументов комбинатора

- Базовый комбинатор $\mathrm{P}\ x_1 x_2 x_3 \ldots x_N = \mathrm{E}$ можно представить λ-термом $\lambda x_1 x_2 x_3 \ldots x_N.E$

Базовый комбинатор (2/2)

- Примеры базовых комбинаторов:

```
I x = x
K xy = x
W xy = xyy
S xyz = xz(yz)
B xyz = x(yz)
C xyz = xzy
```

Формальная система (1/5)

- Четвёрка <S, F, A, R>
  - S – алфавит
  - F – множество выражений (формул)
  - $A \subseteq F$ – аксиомы
  - R – правила вывода

Формальная система (2/5)

- Четвёрка <S, F, A, R>
  - S – алфавит. S = { A..Z, a..z, (,) }
    - A..Z обозначают комбинаторы
    - a..z обозначают переменные

Формальная система (3/5)

- Четвёрка <S, F, A, R>

  - F – множество выражений (формул).

    $F = \{ L=K \mid L, K \in C \}$
  - C – множество комбинаторных термов, строится индуктивно:
    1. Если x – переменная, то $x \in C$;
    2. Если X – комбинатор, то $X \in C$;
    3. Если M и N – термы, то $(MN) \in C$. Семантика (MN) – аппликация, подстановка выражения N в M.
  - Комбинатор – элемент множества C.

Формальная система (4/5)

- Четвёрка <S, F, A, R>

  - R – правила вывода. ∀L, M, N∈C:
    1. M=M
    2. M=N => N=M
    3. M=N, N=L => M=L
    4. M=N => ML=NL
    5. M=N => LM=LN

Формальная система (5/5)

- Четвёрка <S, F, A, R>
  - A⊆F – аксиомы. Самое сложное в формальной системе комбинаторной логики.
  - Комбинаторный базис – набор базовых комбинаторов, составляющий множество аксиом данной формальной системы
    - Набор аксиом можно выбирать удобным образом
    - Выбранный набор должен быть полным. Полный комбинаторный базис – это базис, в котором можно выразить любой комбинатор

Комбинаторный базис (1/6)

- Примеры полных комбинаторных базисов:

1)KS: K xy = x, S xyz = xz(yz)

2)KWB: K xy = x, W xy = xyy, B xyz = x(yz)

3)IBCS:

I x = x

B xyz = x(yz)

C xyz = xzy

S xyz = xz(yz)

4)KBCS:

K xy = x

B xyz = x(yz)

C xyz = xzy

S xyz = xz(yz)

Комбинаторный базис (2/6)

- Как выразить комбинатор в выбранном базисе? Проиллюстрируем на примере
  - Выразим комбинатор I в базисе KS:
    - `K xy = x`
    - `S xyz = xz(yz)`
  - Проанализируем работу комбинатора I с произвольной переменной p:

```
I p = p
    = K p (K p)
    = S K K p
```

  - В силу произвольности выбора p:

```
I = S K K
```

Лирическое отступление 1

- Приведённые комбинаторы столь известны, что имеют свои собственные имена!

  `I x = x` – тождество

  `K xy = x` – канцелятор

  `W xy = xyy` – дубликатор

  `S xyz = xz(yz)` – коннектор

  `B xyz = x(yz)` – композитор

  `C xyz = xzy` – пермутатор

Лирическое отступление 2

- Базовые комбинаторы очень просто выражаются на языке λ-исчисления

  `I x = x` – тождество: I = λx.x

  `K xy = x` – канцелятор: K = λxy.x

  `W xy = xyy` – дубликатор: W = λxy.xyy

  `S xyz = xz(yz)` – коннектор: S = λxyz.xz(yz)

  `B xyz = x(yz)` – композитор: B = λxyz.x(yz)

  `C xyz = xzy` – пермутатор: C = λxyz.xzy

Лирическое отступление 3

- Базовые комбинаторы ещё проще выражаются на языке Haskell!

```
i x = x -- тождество
k x y = x -- канцелятор
w x y = x y y -- дубликатор
s x y z = x z(y z) -- коннектор
b x y z = x(y z) -- композитор
c x y z = x z y -- пермутатор
```

- Всё это – нормальные определения функций на Haskell

Комбинаторный базис (3/6)

- Как выразить комбинатор в выбранном базисе? Проиллюстрируем на примере, используя нотацию λ-исчисления
  - Проверим, что I = SKK:
    - K = λxy.x
    - S = λxyz.xz(yz)
  - Преобразуем SKK:

    SKK = (λxyz.xz(yz)) K K = λz.xz(yz)[y/K][x/K]

    → λz.Kz(Kz)

    → λz.(λxy.x)z((λxy.x)z) = λz.x[y/((λxy.x)z)][x/z]

    → λz.z → λx.x = **I**

Комбинаторный базис (4/6)

- Комбинаторный базис – это набор λ-термов
  - Значит ли это, что можно выразить любой λ-терм через полный комбинаторный базис?

Комбинаторный базис (5/6)

- Комбинаторный базис – это набор λ-термов
  - Значит ли это, что можно выразить любой λ-терм через полный комбинаторный базис?
  - Да, значит!

Комбинаторный базис (6/6)

- Комбинаторный базис – это набор λ-термов
  - Значит ли это, что можно выразить любой λ-терм через полный комбинаторный базис?
  - Да, значит!
  - Шире: полноту комбинаторного базиса можно обосновать одним из двух способов:
    1) Построить трансформацию произвольного λ-терма в данный базис
    2) Выразить в данном базисе другой полный базис

Построение трансформации (1/8)

- Пример: построим трансформационные правила T[] для базиса KS
- Принцип построения трансформации прост
  - Множество λ-термов определяется индуктивно
  - Необходимо для каждого случая из определения λ-терма построить преобразование, выражающее этот λ-терм через комбинаторы из выбранного базиса

Построение трансформации (2/8)

- Пример: построим трансформационные правила T[] для базиса KS
- Определение множества термов Λ:
  - $x \in I, L \in \Lambda \Rightarrow \lambda x.L \in \Lambda$
  - $I \subset \Lambda$
  - $L \in \Lambda \Rightarrow (L) \in \Lambda$
  - $L, K \in \Lambda \Rightarrow (LK) \in \Lambda$

Построение трансформации (3/8)

- Пример: построим трансформационные правила T[] для базиса KS

  - $I \subset \Lambda$.

    $L \in \Lambda \Rightarrow (L) \in \Lambda$. **Правило 1)** $T[x] \rightarrow x$

  - $L, K \in \Lambda \Rightarrow (LK) \in \Lambda$. **Правило 2)** $T[(LK)] \rightarrow (T[L]\ T[K])$

Построение трансформации (4/8)

- Пример: построим трансформационные правила T[] для базиса KS

  - $x \in I$, $L \in \Lambda \Rightarrow \lambda x.L \in \Lambda$. Рассмотрим 4 случая.

  1) $x \notin FV(L)$. Проанализируем:

    - работу $\lambda x.L$ с произвольной переменной p:

      $(\lambda x.L)p \rightarrow L[x/p] = L$

    - работу KT[L] с произвольной переменной p:

      $KT[L]\ p = (\lambda xy.x)\ T[L]\ p \rightarrow (\lambda y.T[L])\ p = T[L]$

  В силу произвольности выбора p:

  Правило 3) $T[\lambda x.L] \rightarrow KT[L]$, $x \notin FV(L)$

Построение трансформации (5/8)

- Пример: построим трансформационные правила T[] для базиса KS

  - x∈I, L∈Λ => λx.L∈Λ. Рассмотрим 4 случая.

  2)L=x. Следовательно, λx.L = λx.x = I = SKK.

  Правило 4) T[λx.x] → SKK

Построение трансформации (6/8)

- Пример: построим трансформационные правила T[] для базиса KS

  - x∈I, L∈Λ => λx.L∈Λ. Рассмотрим 4 случая.

  3)L=λy.M, x∈FV(M). Для перевода в комбинаторный базис вначале трансформируем «внутреннюю» часть λ-терма, потом внешнюю.

  Правило 5) T[λxy.M] → T[λx.T[λy.M]], x∈FV(M)

Построение трансформации (7/8)

- Пример: построим трансформационные правила T[] для базиса KS

  - $x \in I$, $L \in \Lambda \Rightarrow \lambda x.L \in \Lambda$. Рассмотрим 4 случая.

  4)L=(MN). Проанализируем:

  - работу $\lambda x.(MN)$ с произвольной переменной p:
    $(\lambda x.(MN))p \rightarrow M[x/p]N[x/p] =$ **$(\lambda x.M)p\ ((\lambda x.N)p)$**
  - работу $S(\lambda x.M)(\lambda x.N)$ с произвольной переменной p:
    $S(\lambda x.M)(\lambda x.N)p = (\lambda xyz.xz(yz))\ (\lambda x.M)\ (\lambda x.N)\ p$
    $\rightarrow$ **$(\lambda x.M)p\ ((\lambda x.N)p)$**

  Правило 6) $T[\lambda x.(MN)] \rightarrow (ST[\lambda x.M]T[\lambda x.N])$

Построение трансформации (8/8)

- Пример: построим трансформационные правила T[] для базиса KS

  1) $T[x] \rightarrow x$

  2) $T[(LK)] \rightarrow (T[L]\ T[K])$

  3) $T[\lambda x.L] \rightarrow KT[L],\ x \notin FV(L)$

  4) $T[\lambda x.x] \rightarrow SKK$

  5) $T[\lambda xy.M] \rightarrow T[\lambda x.T[\lambda y.M]],\ x \in FV(M)$

  6) $T[\lambda x.(MN)] \rightarrow (ST[\lambda x.M]T[\lambda x.N])$

  => KS – полный базис

Перевод между базисами

- Комбинаторный базис может иметь произвольное число элементов.
  - Базис размерности 1:

    `X x = xKSK`
- Выразим в базисе X базис KS:

  `K xy = (XX)X`

  `S xyz = X(XX)`
  - Проверьте сами!
- Из полноты базиса KS следует полнота X

Тьюринг-полнота

- Х. Карри: комбинаторная логика является тьюринг-полной.
  - Следовательно, на языке комбинаторной логики можно описывать итеративные вычисления. Как?

Неподвижная точка (1/2)

- Неподвижная точка функции y=f(x) – такая $x \in D(f)$, что f(x) = x
  - Для $f(x) = x^2$ – точки 0 и 1
  - Для f(x) = x – вся область определения
- Неподвижная точка комбинатора F – такой комбинатор X, что FX=X
- Теорема о неподвижной точке: $\forall F \exists X: FX=X$

  Доказательство.

  Пусть W=λx.F(xx), X=WW. Тогда:

  X *= WW = (λx.F(xx))W = F(xx)[x/W] = F(WW) =* ***FX****.*

Неподвижная точка (2/2)

- Комбинатор неподвижной точки – такой комбинатор M, что ∀F: MF=F(MF)
  - То есть MF – неподвижная точка F
- Первым комбинатор неподвижной точки в явном виде выразил Хаскелл Карри:

  Y = S (BWB) (BWB)
  - Это знаменитый Y Combinator, или «парадоксальный комбинатор Карри»
  - С его выражением на языке λ-исчисления мы уже знакомы: λf.(λx.f(xx))(λx.f(xx))
  - С его применением для рекурсии – тоже

Использование комбинаторной логики

- Как комбинаторную логику можно использовать в функциональных языках?
  - Есть языки, основой которых является КЛ
  - Есть и ещё одно назначение. Для его демонстрации расширим базис KS.

Базис KBCS (1/4)

- Трансформационные правила T[] для базиса KS
  1) $T[x] \rightarrow x$
  2) $T[(LK)] \rightarrow (T[L]\ T[K])$
  3) $T[\lambda x.L] \rightarrow KT[L],\ x \notin FV(L)$
  4) $T[\lambda x.x] \rightarrow SKK$
  5) $T[\lambda xy.M] \rightarrow T[\lambda x.T[\lambda y.M]],\ x \in FV(M)$
  6) $T[\lambda x.(MN)] \rightarrow (ST[\lambda x.M]T[\lambda x.N])$
- Введём дополнительные условия в правиле 6

Базис KBCS (2/4)

6) T[λx.(MN)]

6.1) Пусть $x \in FV(M)$, $x \in FV(N)$.

$T[\lambda x.(MN)] \rightarrow (S\ T[\lambda x.M]\ T[\lambda x.N])$

6.2) Пусть $x \in FV(M)$, $x \notin FV(N)$.

$T[\lambda x.(MN)] \rightarrow (\mathbf{C}\ T[\lambda x.M]\ \mathbf{T[N]})$

6.3) Пусть $x \notin FV(M)$, $x \in FV(N)$.

$T[\lambda x.(MN)] \rightarrow (\mathbf{B}\ \mathbf{T[M]}\ T[\lambda x.N])$

- Получаем трансформацию в базис KBCS

Базис KBCS (3/4)

- Рассмотрим выражение:

  (λf.f (MLT 5 6))(λn.ADD n n)

  → (λn.ADD n n) (MLT 5 6)

  → ADD (MLT 5 6) (MLT 5 6)

  → ADD 30 (MLT 5 6) → ADD 30 30 → **60**

  - Нормальная редукционная стратегия гарантирует результат, но выполняется заметно дольше, так как дублирует вычисление (MLT 5 6)

Базис KBCS (4/4)

- Переведём выражение в базис KBCS:

  T[(λf.f (MLT 5 6))(λn.ADD n n)]

  (2) → T[(λf.f (MLT 5 6))] T[(λn.ADD n n)]

  (6.2) → C T[λf.f] T[MLT 5 6] T[(λn.ADD n n)]

  (4) → C I T[MLT 5 6] T[(λn.ADD n n)]

  (6.1) → C I T[MLT 5 6] (S T[λn.ADD n] T[λn.n])

  (η) → C I T[MLT 5 6] (S T[ADD] T[λn.n])

  (4) → **C I T[MLT 5 6] (S T[ADD] I)**

  → ...

Лирическое отступление 4

- Как мы смогли перейти от (λn.ADD n) к ADD?

  Правило η-конверсии: λx.Mx => M

- Как обосновать η-конверсию на основе α- и β-редукции?

  Проанализируем работу λx.Mx с произвольной переменной p... Обоснуйте сами!

Лирическое отступление 5

- Как мы смогли перейти от (λn.ADD n) к ADD?

  Правило η-конверсии: λx.Mx => M

- Проиллюстрируем η-конверсию на другом примере: оператор IF

  IF = λcab.cab = λc.(λa.(λb.(ca)b) = λc.(λa.(λb.(ca)b)

  (η) → λc.(λa.ca) = λc.(λa.ca)

  (η) → λc.c = λc.c

  (η) → ...

  - Да, оператор IF – это семантический нуль. Он не выполняет никакой работы и записан только для красоты.

Редукция на графах (1/8)

- Представим выражение

  C I T[MLT 5 6] (S T[ADD] I)

  графически:

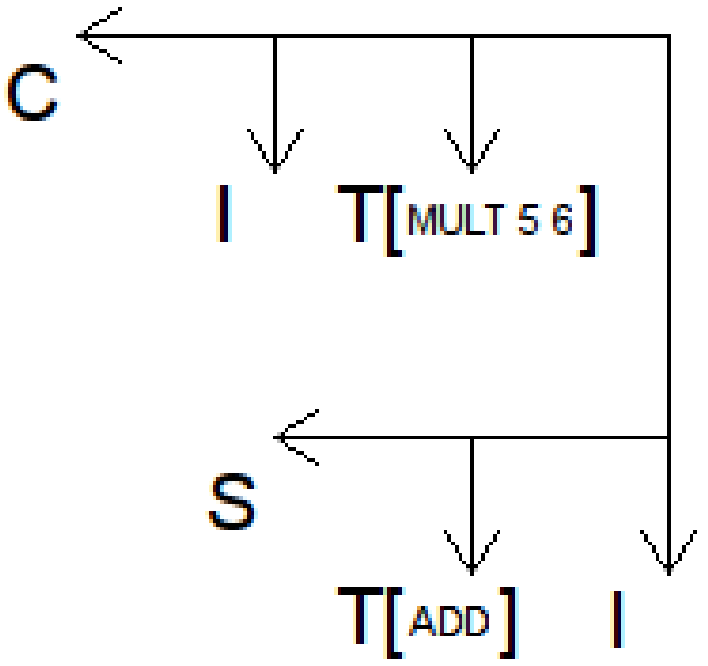
  - Стрелка влево обозначает «вызываемую функцию» – комбинатор, имеющий аргументы
  - Стрелка вниз – аргументы комбинатора

Редукция на графах (2/8)

- Представим выражение

  C I T[MLT 5 6] (S T[ADD] I)

  графически:



Редукция на графах (3/8)

- Представим выражение

  C I T[MLT 5 6] (S T[ADD] I)

  графически.

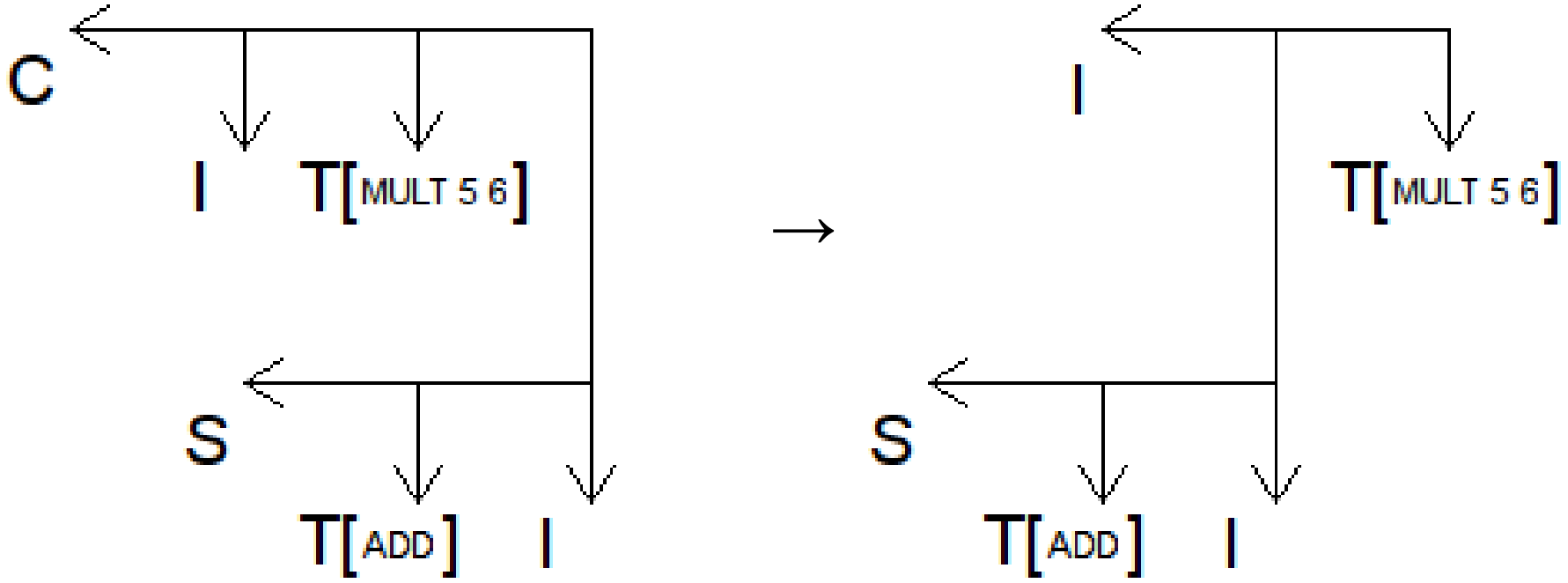
  - Получим граф.
  - Преобразуем граф согласно правилам вычисления комбинаторных выражений

Редукция на графах (4/8)

- Преобразуем граф согласно правилам вычисления комбинаторных выражений.

C I T[MLT 5 6] (S T[ADD] I)

→ I (S T[ADD] I) T[MLT 5 6]

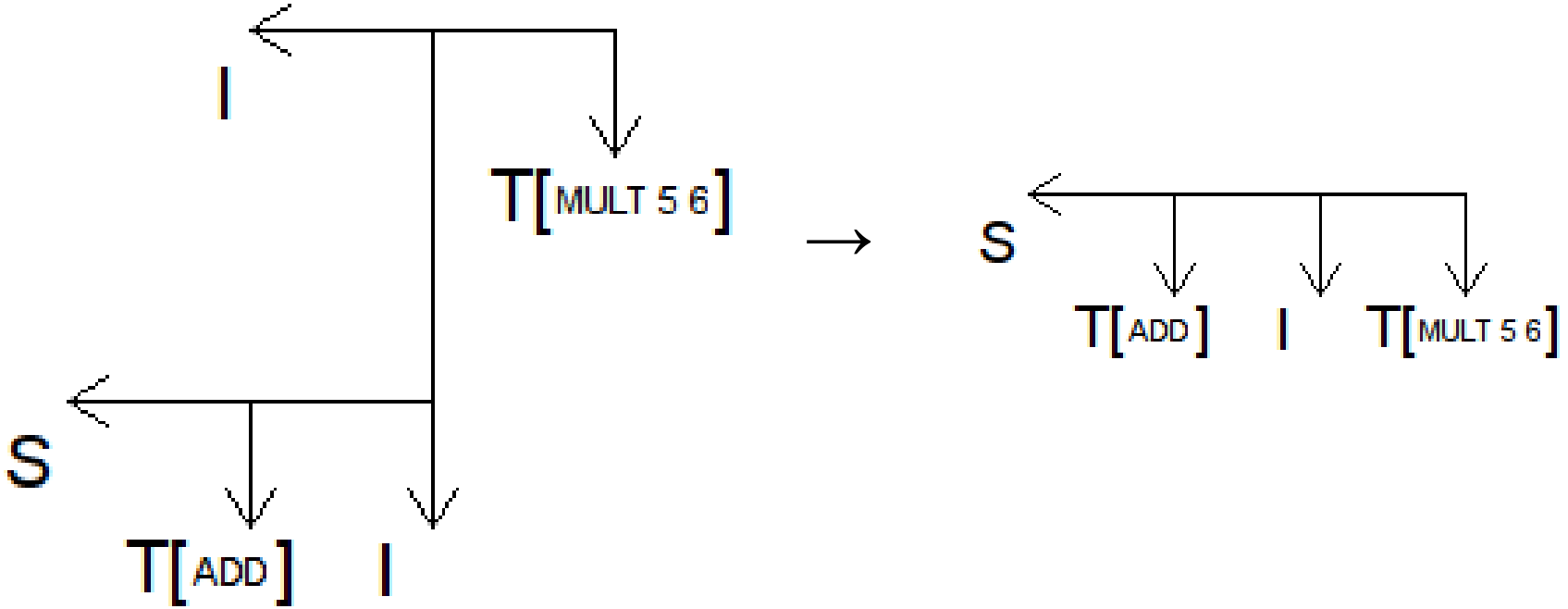


Редукция на графах (5/8)

- Преобразуем граф согласно правилам вычисления комбинаторных выражений.

I (S T[ADD] I) T[MLT 5 6]

→ S T[ADD] I T[MLT 5 6]



Редукция на графах (6/8)

- Преобразуем граф согласно правилам вычисления комбинаторных выражений.

  S T[ADD] I T[MLT 5 6]

  → **T[ADD] T[MLT 5 6] (I T[MLT 5 6])**

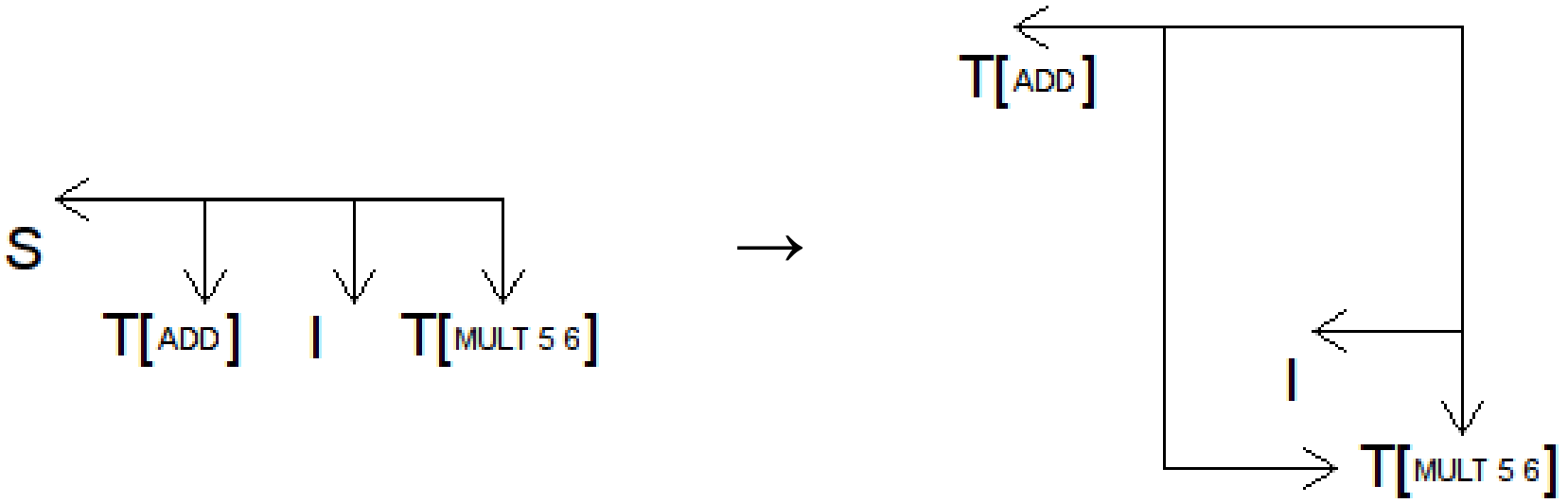
  - Здесь выражение T[MLT 5 6] дублируется – благодаря свойству комбинатора S
  - Мы заранее знаем про это свойство, поэтому можем обработать не дублирование целого объекта, а дублирование ссылки на объект

Редукция на графах (7/8)

- Преобразуем граф согласно правилам вычисления комбинаторных выражений.

  S T[ADD] I T[MLT 5 6]

  → T[ADD] T[MLT 5 6] (I T[MLT 5 6])

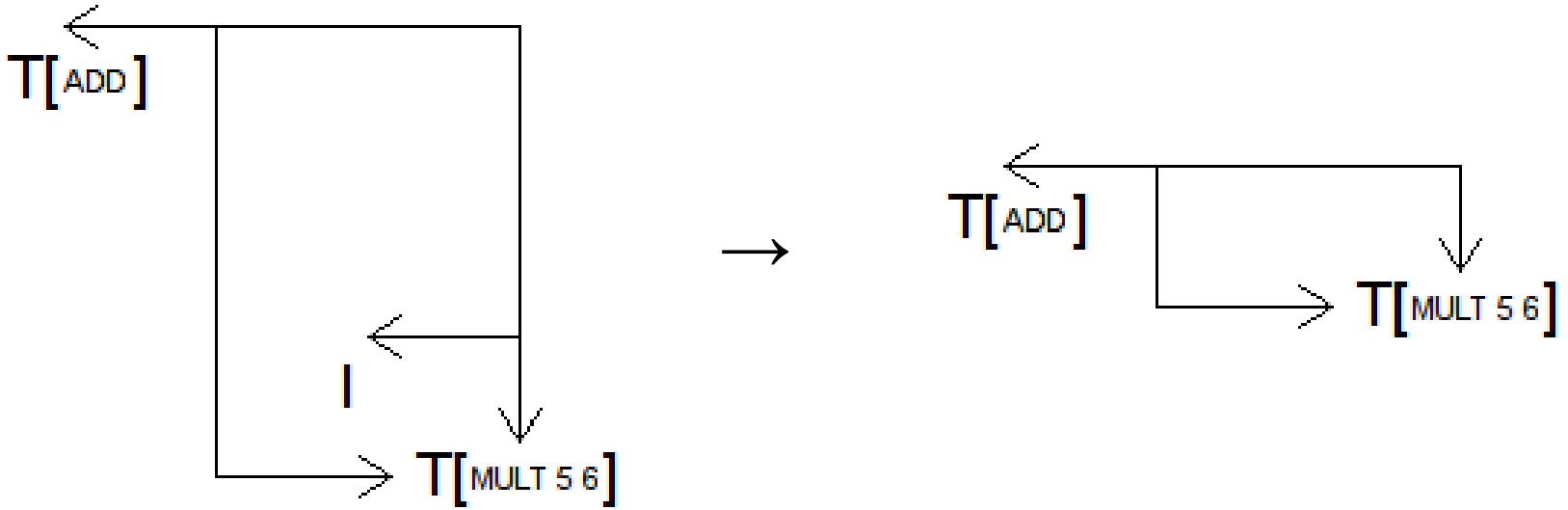


Редукция на графах (8/8)

- Преобразуем граф согласно правилам вычисления комбинаторных выражений.

  T[ADD] T[MLT 5 6] (I T[MLT 5 6])

  → T[ADD] T[MLT 5 6] T[MLT 5 6]

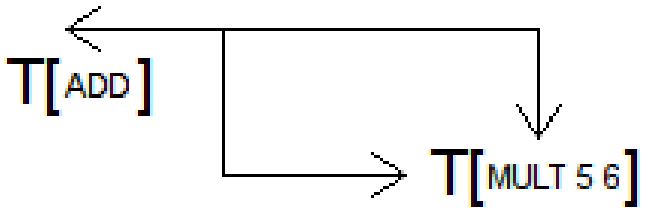


Ленивая редукционная стратегия (1/4)

- Преобразуем граф согласно правилам вычисления комбинаторных выражений.

  C I T[MLT 5 6] (S T[ADD] I)

  → **T[ADD] T[MLT 5 6] T[MLT 5 6]**



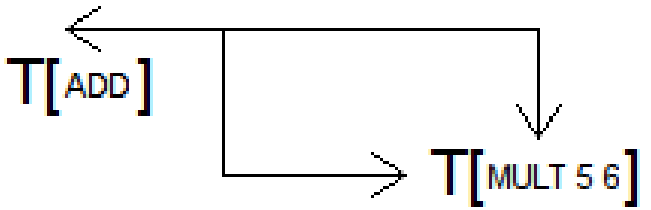
  - В ходе преобразований мы (почти) соблюдали семантику нормальной редукционной стратегии
  - Но дублирования объектов не произошло

Ленивая редукционная стратегия (2/4)

- Преобразуем граф согласно правилам вычисления комбинаторных выражений.

  C I T[MLT 5 6] (S T[ADD] I)

  → **T[ADD] T[MLT 5 6] T[MLT 5 6]**



  - Выражение T[MLT 5 6] будет вычислено только один раз и только тогда, когда потребуется его значение

Ленивая редукционная стратегия (3/4)

- Такой метод редукции носит название «ленивая редукционная стратегия»
  - ЛРС соблюдает семантику НРС, поэтому всегда находит нормальную форму, если таковая существует
  - ЛРС вычисляет выражение только один раз, поэтому работает со скоростью, близкой к АРС
  - ЛРС вычисляет выражение только тогда, когда потребуется его значение (поэтому «ленивая»)
  - Плата за всё это – увеличение занимаемого объёма памяти для хранения дерева

Ленивая редукционная стратегия (4/4)

- Такой метод редукции носит названия:
  - «ленивая редукционная стратегия»
  - «отложенные вычисления»
  - «ленивые вычисления»
- Используя его, можно определять бесконечные структуры данных

  Пример – бесконечный список. В АРС попытка получить бесконечный список привела бы к зацикливанию. В ЛРС такой список – просто элемент вычислительного дерева, который будет «раскручен», когда потребуются элементы списка

Ещё 5 слов о трансляторе

- Трансформируя λ-выражение λf.f (MLT 5 6)) (λn.ADD n n), мы не стали трансформировать λ-выражение MLT 5 6. Почему?

Ещё 4 слова о трансляторе

- Трансформируя λ-выражение λf.f (MLT 5 6)) (λn.ADD n n), мы не стали трансформировать λ-выражение MLT 5 6. Почему?
- Современные трансляторы функциональных языков переводят функциональный код в машинный, предназначенный для выполнения на массово выпускаемых процессорах, например, семейства Intel x86.
- В этих процессорах уже есть встроенная арифметика. Эффективнее использовать её, нежели арифметику λ-исчисления

Ещё 3 слова о трансляторе

- То же самое верно для ряда других концепций. В том числе для рекурсивных вычислений.
- Рекурсию можно реализовать на основе редукции графа комбинатора Y, но эффективнее пользоваться встроенными в процессор примитивами для вызова функций, в том числе рекурсивного

Ещё 2 слова о трансляторе

- То же самое верно для ряда других концепций. В том числе для рекурсивных вычислений.
- Рекурсию можно реализовать на основе редукции графа комбинатора Y, но эффективнее пользоваться встроенными в процессор примитивами для вызова функций, в том числе рекурсивного
- Проблема: в Intel x86 рекурсивный вызов не может работать бесконечно!

Ещё 1 слово о трансляторе

```
$ cat > rec.c << EOF
> int f(int i) { return f(i); }
> int main(void) { return f(3); }
> EOF
$ gcc rec.c
$ ./a.out
Segmentation fault
$
```

Хвостовая рекурсия (1/7)

- Проблема: в Intel x86 рекурсивный вызов не может работать бесконечно!
  - Это оттого, что при вызове функции её аргументы (а также адрес возврата) кладутся на стек, в так называемый стековый кадр (stack frame). N вложенных вызовов – N стековых кадров. А стек не бесконечен
  - Подробности – в курсе «Архитектура ЭВМ и язык ассемблера»

Хвостовая рекурсия (2/7)

- Проблема: в Intel x86 рекурсивный вызов не может работать бесконечно!
- Решение: для большого класса рекурсивных вызовов есть возможность не создавать новый кадр, а модифицировать старый
- Такой приём носит название «хвостовая рекурсия» («tail recursion», «tail call»)

Хвостовая рекурсия (3/7)

- Хвостовая рекурсия требует поддержки со стороны программиста
  - Для того, чтобы вместо создания нового стекового кадра можно было модифицировать старый, требуется, чтобы возвращаемым значением вызываемой рекурсивно функции было значение (этой же или другой) функции.
  - Это возвращаемое значение не должно модифицироваться

Хвостовая рекурсия (4/7)

- Хвостовая рекурсия требует поддержки со стороны программиста. Пример:

```
int f(int i) { return f(i); }
```

- Это хвостовая рекурсия
  - Правда, на C, так что всё равно не заработает
  - Впрочем, спустя 30 лет после появления хвостовая рекурсия стала появляться и в императивных языках. Например, компилятор gcc для этого имеет опцию оптимизации -foptimize-sibling-calls, включённую по умолчанию на уровнях оптимизации -O2 и -O3. Попробуйте

Хвостовая рекурсия (5/7)

- Хвостовая рекурсия требует поддержки со стороны программиста. Пример:

```
int f(int i) { return g(i); }
int g(int i) { return f(i); }
```

- Это хвостовая рекурсия.

Хвостовая рекурсия (6/7)

- Хвостовая рекурсия требует поддержки со стороны программиста. Пример:

```
int fact(int n) {
    if (n == 1) return 1;
    else return n*fact(n-1);
}
```

- Это не хвостовая рекурсия. Возвращаемое значение fact() – n*fact(n-1), это не функция

Хвостовая рекурсия (7/7)

- Хвостовая рекурсия требует поддержки со стороны программиста. Пример:

```
int fact_tail(int n, int k) {
    if (n == 1) return k;
    else return fact_tail(n-1, k*n);
}
int fact(int n)
{ return fact_tail(n, 1); }
```

- Это хвостовая рекурсия. Параметр k часто называют «накапливающим параметром»

Лирическое отступление 6 (1/5)

- То же самое на Haskell:

```
fact :: Int -> Int
fact n = fact_tail n 1

fact_tail :: Int -> Int -> Int
fact_tail n k =   if n == 1 then k
                  else fact_tail (n-1) k*n
```

Лирическое отступление 6 (2/5)

- То же самое на Haskell:

```
fact :: Int -> Int
fact n = fact_tail n 1

fact_tail :: Int -> Int -> Int
fact_tail n k       | n == 1    = k
                    | otherwise
                      = fact_tail (n-1) k*n
```

(пробелы расставлены для красоты)

Лирическое отступление 6 (3/5)

- То же самое на Haskell:

```
fact :: Int -> Int
fact n = fact_tail n 1

fact_tail :: Int -> Int -> Int
fact_tail n k = case n == 1 of
    True              -> k
    False             -> fact_tail (n-1) k*n
```

Лирическое отступление 6 (4/5)

- То же самое на Haskell:

```
fact :: Int -> Int
fact n = fact_tail n 1

fact_tail :: Int -> Int -> Int
fact_tail n k = case n      of
           1          -> k
           n          -> fact_tail (n-1) k*n
```

Лирическое отступление 6 (5/5)

- То же самое на Haskell:

```
fact :: Int -> Int
fact n = fact_tail n 1

fact_tail :: Int -> Int -> Int

fact_tail 1 k          = k
fact_tail n k          = fact_tail (n-1) k*n
```

Лирическое отступление 7

- То же самое на Haskell можно записать ещё сотней способов.

  «С монадными трансформерами этот алгоритм написать тоже можно, просто я так не пишу».

  - Дмитрий Астапов

- Язык Haskell позволяет программисту выработать свой собственный стиль. Но об этом в другой раз