

Изоморфизм Карри-Говарда

Основы функционального программирования

Михаил Беляев Вадим Цесько

Санкт-Петербургский государственный политехнический университет

16 декабря 2015 г.

Содержание

1 Введение

Содержание

- 1 Введение
- 2 Идея

Содержание

- 1 Введение
- 2 Идея
- 3 Логика и исчисление комбинаторов

Содержание

- 1 Введение
- 2 Идея
- 3 Логика и исчисление комбинаторов
- 4 Расширения системы

Содержание

- 1 Введение
- 2 Идея
- 3 Логика и исчисление комбинаторов
- 4 Расширения системы
- 5 Выводы

- 1934 г. — *Haskell Curry*:

¹Комбинатор — это функция высшего порядка, которая использует только применение функций и ранее определённые комбинаторы.

- 1934 г. — *Haskell Curry*:
 - Типы комбинаторов ¹ можно рассматривать как схемы аксиом для импликативной логики

¹Комбинатор — это функция высшего порядка, которая использует только применение функций и ранее определённые комбинаторы

- 1934 г. — *Haskell Curry*:
 - Типы комбинаторов ¹ можно рассматривать как схемы аксиом для импликативной логики
- 1958 г. — *Haskell Curry*:

¹Комбинатор — это функция высшего порядка, которая использует только применение функций и ранее определённые комбинаторы

- 1934 г. — *Haskell Curry*:
 - Типы комбинаторов ¹ можно рассматривать как схемы аксиом для импликативной логики
- 1958 г. — *Haskell Curry*:
 - Система доказательства Гильберта частично совпадает с моделью вычислений логики комбинаторов

¹Комбинатор — это функция высшего порядка, которая использует только применение функций и ранее определённые комбинаторы

- 1934 г. — *Haskell Curry*:
 - Типы комбинаторов ¹ можно рассматривать как схемы аксиом для импликативной логики
- 1958 г. — *Haskell Curry*:
 - Система доказательства Гильберта частично совпадает с моделью вычислений логики комбинаторов
- 1969 г. — *William Alvin Howard*:

¹Комбинатор — это функция высшего порядка, которая использует только применение функций и ранее определённые комбинаторы

- 1934 г. — *Haskell Curry*:
 - Типы комбинаторов ¹ можно рассматривать как схемы аксиом для импликативной логики
- 1958 г. — *Haskell Curry*:
 - Система доказательства Гильберта частично совпадает с моделью вычислений логики комбинаторов
- 1969 г. — *William Alvin Howard*:
 - Более высокоуровневая система доказательств — «Естественная дедукция», — отображается в типизированное λ -исчисление

¹Комбинатор — это функция высшего порядка, которая использует только применение функций и ранее определённые комбинаторы

Изоморфизм Карри-Говарда

Системы доказательств и модели вычислений суть **одинаковые** по структуре типы объектов

Неформально

Любой **типизированной** системе соответствует некоторая логика, и наоборот

- Доказательство — программа

Интерпретации

- Доказательство — программа
- Доказываемая формула — тип программы

Интерпретации

- **Доказательство** — программа
 - Доказываемая **формула** — тип программы
-
- **Тип** возвращаемого значения функции — **теорема** в логике

Интерпретации

- **Доказательство** — программа
 - Доказываемая **формула** — тип программы
-
- **Тип** возвращаемого значения функции — **теорема** в логике
 - **Типы** аргументов — **гипотезы**

Интерпретации

- **Доказательство** — программа
 - Доказываемая **формула** — тип программы
-
- **Тип** возвращаемого значения функции — **теорема** в логике
 - **Типы** аргументов — **гипотезы**
 - **Программа**, вычисляющая функцию — **доказательство** теоремы

Последствия

- Новый класс формальных систем — одновременно и системы доказательств, и типизированные функциональные языки программирования (см. Coq)
- Основа современной теории типов
- Новые вопросы в теории вычислений (соответствия других систем доказательств и моделей вычислений)
- Унификация математической логики и основ Computer Science

Соглашения

- Чисто функциональные языки программирования (e. g. Haskell)

Соглашения

- Чисто функциональные языки программирования (e. g. Haskell)
- Не совсем формально

Соглашения

- Чисто функциональные языки программирования (e. g. Haskell)
- Не совсем формально
- Вы сдали зачёт по курсу «Языки программирования» 😊

Соглашения

- Чисто функциональные языки программирования (e. g. Haskell)
- Не совсем формально
- Вы сдали зачёт по курсу «Языки программирования» 😊
- Вы помните предыдущую лекцию 😊

Будем всё показывать на примере Haskell

Но:

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)
- Никакой рекурсии в коде!

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)
- Никакой рекурсии в коде!
- Никакой рекурсии в типах!

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)
- Никакой рекурсии в коде!
- Никакой рекурсии в типах!
- Запрещены функции `undefined` и `error`

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)
- Никакой рекурсии в коде!
- Никакой рекурсии в типах!
- Запрещены функции `undefined` и `error`
- Не используются стандартные типы (кроме некоторых, см. далее)

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)
- Никакой рекурсии в коде!
- Никакой рекурсии в типах!
- Запрещены функции `undefined` и `error`
- Не используются стандартные типы (кроме некоторых, см. далее)
- Будем называть типы большими буквами: `A`, `B`, `C` ...

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)
- Никакой рекурсии в коде!
- Никакой рекурсии в типах!
- Запрещены функции `undefined` и `error`
- Не используются стандартные типы (кроме некоторых, см. далее)
- Будем называть типы большими буквами: `A`, `B`, `C` ...
- Из зависимых типов — только кортежи и *вариантные типы*

Будем всё показывать на примере Haskell

Но:

- Только чистые функции (никаких монад!)
- Никакой рекурсии в коде!
- Никакой рекурсии в типах!
- Запрещены функции `undefined` и `error`
- Не используются стандартные типы (кроме некоторых, см. далее)
- Будем называть типы большими буквами: `A`, `B`, `C` ...
- Из зависимых типов — только кортежи и *вариантные типы*
 - Вместо `data A = A b B | A c C` будем писать `B | C`

Для тех, кто в глаза не видел Haskell
(кстати, кто-нибудь видел?)

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Имена типов начинаются с большой буквы, имена функций — с маленькой

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Имена типов начинаются с большой буквы, имена функций — с маленькой
- Переменные над типами тоже начинаются с маленьких букв

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Имена типов начинаются с большой буквы, имена функций — с маленькой
- Переменные над типами тоже начинаются с маленьких букв
- Кортежи: $(x, y) :: (A, B)$, $(x, y, z) :: (A, B, C)$,
 $(2, x, \text{True}) :: (\text{Int}, a, \text{Bool})$
- Unit: $()$

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Имена типов начинаются с большой буквы, имена функций — с маленькой
- Переменные над типами тоже начинаются с маленьких букв
- Кортежи: $(x, y) :: (A, B)$, $(x, y, z) :: (A, B, C)$,
 $(2, x, \text{True}) :: (\text{Int}, a, \text{Bool})$
- Unit: $()$
- Функциональные типы: $a \rightarrow a$, $\text{Int} \rightarrow \text{Bool}$,
 $(a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Имена типов начинаются с большой буквы, имена функций — с маленькой
- Переменные над типами тоже начинаются с маленьких букв
- Кортежи: $(x,y) :: (A,B)$, $(x,y,z) :: (A,B,C)$,
 $(2,x, \text{True}) :: (\text{Int}, a, \text{Bool})$
- Unit: $()$
- Функциональные типы: $a \rightarrow a$, $\text{Int} \rightarrow \text{Bool}$,
 $(a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
- Синтаксис определения функции:

```
-- const takes 2 parameters and
-- returns the first one
const :: a -> b -> a
const x y = x
```

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Имена типов начинаются с большой буквы, имена функций — с маленькой
- Переменные над типами тоже начинаются с маленьких букв
- Кортежи: $(x,y) :: (A,B)$, $(x,y,z) :: (A,B,C)$,
 $(2,x, \text{True}) :: (\text{Int}, a, \text{Bool})$
- Unit: $()$
- Функциональные типы: $a \rightarrow a$, $\text{Int} \rightarrow \text{Bool}$,
 $(a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
- Синтаксис определения функции:

```
-- const takes 2 parameters and
-- returns the first one
const :: a -> b -> a
const x y = x
```

- Всегда используется *каррирование*: $a \rightarrow b \rightarrow c$ это то же самое, что и $a \rightarrow (b \rightarrow c)$

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Типы в Haskell являются вариантными (datatypes):

```
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b
```

Для тех, кто в глаза не видел Haskell (кстати, кто-нибудь видел?)

- Типы в Haskell являются вариантными (datatypes):

```
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b
```

- Для наших целей это слишком сложно, мы будем писать просто `a|b`, `Int|Bool`

Доказательство теорем

Доказательство теорем

- 1 Формулируем теорему

Доказательство теорем

- 1 Формулируем теорему
- 2 Конструируем тип, который выражает эту теорему

Доказательство теорем

- 1 Формулируем теорему
- 2 Конструируем тип, который выражает эту теорему
- 3 Находим значение, имеющее этот тип

Высказывания суть типы

Высказывания суть типы

- Что означает тип $A \rightarrow B$?

Высказывания суть типы

- Что означает тип $A \rightarrow B$?
- $A \rightarrow B$

Высказывания суть типы

- Что означает тип $A \rightarrow B$?
- $A \rightarrow B$
- Если типы A и B можно интерпретировать логически

Высказывания суть типы

- Что означает тип $A \rightarrow B$?
- $A \rightarrow B$
- Если типы A и B можно интерпретировать логически
- $A \rightarrow B$, если $A \rightarrow B$ — населённый (inhabited) тип

Высказывания суть типы

- Что означает тип $A \rightarrow B$?
- $A \rightarrow B$
- Если типы A и B можно интерпретировать логически
- $A \rightarrow B$, если $A \rightarrow B$ — населённый (inhabited) тип

Пример

- Haskell-функция `const` :: $a \rightarrow b \rightarrow a$
- В логике транслируется в $a \rightarrow (b \rightarrow a)$
- Это теорема, поскольку тип $a \rightarrow b \rightarrow a$ населён значением `const`

Логические операции

- Оператор импликации \rightarrow в логике соответствует конструктору типа \rightarrow

Логические операции

- Оператор импликации \rightarrow в логике соответствует конструктору типа \rightarrow
- Рассмотрим: \wedge , \vee , *true*, *false*, \neg

Конъюнкция и дизъюнкция

- $A \wedge B$ — теорема, значит:
 - A — теорема
 - B — теорема

Конъюнкция и дизъюнкция

- $A \wedge B$ — теорема, значит:
 - A — теорема
 - B — теорема
- Нужно найти значение (A, B) :
 - тип A соответствует A
 - тип B соответствует B

Конъюнкция и дизъюнкция

- $A \wedge B$ — теорема, значит:
 - A — теорема
 - B — теорема
- Нужно найти значение (A, B) :
 - тип A соответствует A
 - тип B соответствует B
- Аналогично высказыванию $A \vee B$ соответствует тип $A \mid B$

Значение true

- Нужен тип, который всегда населён

Значение true

- Нужен тип, который всегда населён
- В Haskell его очень просто написать:
 - `data Unit = Unit`

Значение true

- Нужен тип, который всегда населён
- В Haskell его очень просто написать:
 - `data Unit = Unit`
- Так же сошёлся бы любой стандартный тип, имеющий значения (`Char`, `Int`, `String...`)

Значение true

- Нужен тип, который всегда населён
- В Haskell его очень просто написать:
 - `data Unit = Unit`
- Так же сошёлся бы любой стандартный тип, имеющий значения (`Char`, `Int`, `String...`)
- Наиболее простым таким типом является тип `()`

Значение true

- Нужен тип, который всегда населён
- В Haskell его очень просто написать:
 - `data Unit = Unit`
- Так же сошёлся бы любой стандартный тип, имеющий значения (`Char`, `Int`, `String...`)
- Наиболее простым таким типом является тип `()`

Следствия:

Значение `true`

- Нужен тип, который всегда населён
- В Haskell его очень просто написать:
 - `data Unit = Unit`
- Так же сгодился бы любой стандартный тип, имеющий значения (`Char`, `Int`, `String...`)
- Наиболее простым таким типом является тип `()`

Следствия:

- $((() | A) \text{ и } (A | ()))$ населены для любого типа A :
 - Т.е. $true \vee A$ и $A \vee true$ теоремы

Значение `true`

- Нужен тип, который всегда населён
- В Haskell его очень просто написать:
 - `data Unit = Unit`
- Так же сошёлся бы любой стандартный тип, имеющий значения (`Char`, `Int`, `String...`)
- Наиболее простым таким типом является тип `()`

Следствия:

- $(() \mid A)$ и $(A \mid ())$ населены для любого типа A :
 - Т.е. $true \vee A$ и $A \vee true$ теоремы
- $((), A)$ и $(A, ())$ ведут себя так же как тип A :
 - Соответствуют $true \wedge A$ и $A \wedge true$

Значение true

- Нужен тип, который всегда населён
- В Haskell его очень просто написать:
 - `data Unit = Unit`
- Так же сгодился бы любой стандартный тип, имеющий значения (`Char`, `Int`, `String...`)
- Наиболее простым таким типом является тип `()`

Следствия:

- $(() \mid A)$ и $(A \mid ())$ населены для любого типа A :
 - Т.е. $true \vee A$ и $A \vee true$ теоремы
- $((), A)$ и $(A, ())$ ведут себя так же как тип A :
 - Соответствуют $true \wedge A$ и $A \wedge true$
- $A \rightarrow ()$ всегда населён (функцией `unit x = ()`):
 - $A \rightarrow true$ всегда верно

Значение `false`

- Нужен ненаселённый тип (не путаем с типом `()`)

Значение false

- Нужен ненаселённый тип (не путаем с типом `()`)
- В GHC есть «фантомный» тип \perp , мы будем использовать обозначение `Bot`

Значение false

- Нужен ненаселённый тип (не путаем с типом `()`)
- В GHC есть «фантомный» тип \perp , мы будем использовать обозначение `Bot`
 - У него нет значений
 - Его можно привести к любому другому типу: `cast :: Bot -> a`
 - Именно такой тип *на самом деле* имеет функция `undefined`

Значение false

- Нужен ненаселённый тип (не путаем с типом `()`)
- В GHC есть «фантомный» тип \perp , мы будем использовать обозначение `Bot`
 - У него нет значений
 - Его можно привести к любому другому типу: `cast :: Bot -> a`
 - Именно такой тип *на самом деле* имеет функция `undefined`

Следствия:

Значение `false`

- Нужен ненаселённый тип (не путаем с типом `()`)
- В GHC есть «фантомный» тип \perp , мы будем использовать обозначение `Bot`
 - У него нет значений
 - Его можно привести к любому другому типу: `cast :: Bot -> a`
 - Именно такой тип *на самом деле* имеет функция `undefined`

Следствия:

- (Bot, A) и (A, Bot) ненаселены для любого типа A :
 - Т.е. $\text{false} \wedge A$ и $A \wedge \text{false}$ не теоремы

Значение `false`

- Нужен ненаселённый тип (не путаем с типом `()`)
- В GHC есть «фантомный» тип \perp , мы будем использовать обозначение `Bot`
 - У него нет значений
 - Его можно привести к любому другому типу: `cast :: Bot -> a`
 - Именно такой тип *на самом деле* имеет функция `undefined`

Следствия:

- (Bot, A) и (A, Bot) ненаселены для любого типа A :
 - Т.е. $\text{false} \wedge A$ и $A \wedge \text{false}$ не теоремы
- $(\text{Bot} \mid A)$ и $(A \mid \text{Bot})$ ведут себя так же как тип A :
 - Соответствуют $\text{false} \vee A$ и $A \vee \text{false}$

Значение `false`

- Нужен ненаселённый тип (не путаем с типом `()`)
- В GHC есть «фантомный» тип \perp , мы будем использовать обозначение `Bot`
 - У него нет значений
 - Его можно привести к любому другому типу: `cast :: Bot -> a`
 - Именно такой тип *на самом деле* имеет функция `undefined`

Следствия:

- (Bot, A) и (A, Bot) ненаселены для любого типа A :
 - Т.е. $\text{false} \wedge A$ и $A \wedge \text{false}$ не теоремы
- $(\text{Bot} \mid A)$ и $(A \mid \text{Bot})$ ведут себя так же как тип A :
 - Соответствуют $\text{false} \vee A$ и $A \vee \text{false}$
- $\text{Bot} \rightarrow A$ населён всегда (функцией `cast`):
 - $\text{false} \rightarrow A$ всегда верно

Отрицание

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

- Операция \neg превращает теоремы в нетеоремы и наоборот

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

- Операция \neg превращает теоремы в нетеоремы и наоборот
- В Haskell синоним типа: `type Not a = a -> Bot`

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

- Операция \neg превращает теоремы в нетеоремы и наоборот
- В Haskell синоним типа: `type Not a = a -> Bot`
- Если A — тип-теорема, то тип $A -> Bot$ ненаселён

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

- Операция \neg превращает теоремы в нетеоремы и наоборот
- В Haskell синоним типа: `type Not a = a -> Bot`
- Если A — тип-теорема, то тип $A \rightarrow Bot$ ненаселён
- Если A — не теорема, то:

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

- Операция \neg превращает теоремы в нетеоремы и наоборот
- В Haskell синоним типа: `type Not a = a -> Bot`
- Если A — тип-теорема, то тип $A \rightarrow Bot$ ненаселён
- Если A — не теорема, то:
 - A можно заменить на Bot

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

- Операция \neg превращает теоремы в нетеоремы и наоборот
- В Haskell синоним типа: `type Not a = a -> Bot`
- Если A — тип-теорема, то тип $A \rightarrow Bot$ ненаселён
- Если A — не теорема, то:
 - A можно заменить на `Bot`
 - Функция `id :: Bot -> Bot` населяет `Not A`

Вопрос на засыпку

Можно ли выразить отрицание через уже имеющиеся у нас операции?

- Операция \neg превращает теоремы в нетеоремы и наоборот
- В Haskell синоним типа: `type Not a = a -> Bot`
- Если A — тип-теорема, то тип $A \rightarrow Bot$ ненаселён
- Если A — не теорема, то:
 - A можно заменить на `Bot`
 - Функция `id :: Bot -> Bot` населяет `Not A`
 - Следовательно, `Not A` — тип-теорема

Система доказательств

Аксиомы:

- $A \rightarrow (B \rightarrow A)$
- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

Правило вывода:

- **Modus Ponens**

Система доказательств

Аксиомы:

- $A \rightarrow (B \rightarrow A)$
- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

Правило вывода:

- **Modus Ponens**

Theorem

$A \rightarrow A$

Доказательство.

- **Ax2**[A / C]: $(A \rightarrow (B \rightarrow A)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow A))$
- **Ax1**: $(A \rightarrow B) \rightarrow (A \rightarrow A)$
- **[A → C / B]**: $(A \rightarrow (C \rightarrow A)) \rightarrow (A \rightarrow A)$
- **Ax1**: $A \rightarrow A$



Исчисление комбинаторов

Аксиомы:

Исчисление комбинаторов

Аксиомы:

- $A \rightarrow (B \rightarrow A)$

$$k :: a \rightarrow b \rightarrow a$$

Исчисление комбинаторов

Аксиомы:

- $A \rightarrow (B \rightarrow A)$

$$k :: a \rightarrow b \rightarrow a$$

$$k\ x\ y = x$$

Исчисление комбинаторов

Аксиомы:

- $A \rightarrow (B \rightarrow A)$

$$k :: a \rightarrow b \rightarrow a$$

$$k \ x \ y = x$$

- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

$$s :: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

Исчисление комбинаторов

Аксиомы:

- $A \rightarrow (B \rightarrow A)$

$$k :: a \rightarrow b \rightarrow a$$

$$k \ x \ y = x$$

- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

$$s :: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

$$s \ x \ y \ z = x \ z \ (y \ z)$$

Исчисление комбинаторов

Аксиомы:

- $A \rightarrow (B \rightarrow A)$

$$k :: a \rightarrow b \rightarrow a$$

$$k\ x\ y = x$$

- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

$$s :: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

$$s\ x\ y\ z = x\ z\ (y\ z)$$

Правило вывода:

- **Modus Ponens** Применение функции к аргументу

Исчисление комбинаторов

Аксиомы:

- $A \rightarrow (B \rightarrow A)$

$$k :: a \rightarrow b \rightarrow a$$

$$k \ x \ y = x$$

- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

$$s :: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

$$s \ x \ y \ z = x \ z \ (y \ z)$$

Правило вывода:

- **Modus Ponens** Применение функции к аргументу

Theorem

$$A \rightarrow A \ i :: a \rightarrow a$$

Исчисление комбинаторов

Аксиомы:

- $A \rightarrow (B \rightarrow A)$

$$k :: a \rightarrow b \rightarrow a$$

$$k \ x \ y = x$$

- $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

$$s :: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

$$s \ x \ y \ z = x \ z \ (y \ z)$$

Правило вывода:

- **Modus Ponens** Применение функции к аргументу

Theorem

$$A \rightarrow A \ i :: a \rightarrow a$$

Доказательство

$$i = s \ k \ k$$

Исчисление комбинаторов

- Мы только что изобрели комбинаторную логику! (Combinatory logic)

Исчисление комбинаторов

- Мы только что изобрели комбинаторную логику! (Combinatory logic)
- Подобно тому, как на двух указанных аксиомах можно построить систему доказательств, на функциях s и k можно построить целое исчисление

Исчисление комбинаторов

- Мы только что изобрели комбинаторную логику! (Combinatory logic)
- Подобно тому, как на двух указанных аксиомах можно построить систему доказательств, на функциях s и k можно построить целое исчисление
- Комбинаторное исчисление аналогично типизированному λ -исчислению, только несколько проще

Исчисление комбинаторов

- Мы только что изобрели комбинаторную логику! (Combinatory logic)
- Подобно тому, как на двух указанных аксиомах можно построить систему доказательств, на функциях s и k можно построить целое исчисление
- Комбинаторное исчисление аналогично типизированному λ -исчислению, только несколько проще
- Выведенный нами базис – только один из возможных базисов комбинаторного исчисления, но он вполне полный

Интуиционистская vs классическая логики

- Выше была интуиционистская логика

Интуиционистская vs классическая логики

- Выше была интуиционистская логика
- Теорема классической логики: $\neg(\neg A) \rightarrow A$

Интуиционистская vs классическая логики

- Выше была интуиционистская логика
- Теорема классической логики: $\neg(\neg A) \rightarrow A$
- Это преобразуется в $((A \rightarrow \text{Bot}) \rightarrow \text{Bot})$

Интуиционистская vs классическая логики

- Выше была интуиционистская логика
- Теорема классической логики: $\neg(\neg A) \rightarrow A$
- Это преобразуется в $((A \rightarrow \text{Bot}) \rightarrow \text{Bot})$
- По функции типа $(A \rightarrow \text{Bot}) \rightarrow \text{Bot}$ нужно получить значение типа A

Интуиционистская vs классическая логики

- Выше была интуиционистская логика
- Теорема классической логики: $\neg(\neg A) \rightarrow A$
- Это преобразуется в $((A \rightarrow \text{Bot}) \rightarrow \text{Bot})$
- По функции типа $(A \rightarrow \text{Bot}) \rightarrow \text{Bot}$ нужно получить значение типа A
- Функция типа $(A \rightarrow \text{Bot}) \rightarrow \text{Bot}$ существует, если тип $A \rightarrow \text{Bot}$ ненаселён

Интуиционистская vs классическая логики

- Выше была интуиционистская логика
- Теорема классической логики: $\neg(\neg A) \rightarrow A$
- Это преобразуется в $((A \rightarrow \text{Bot}) \rightarrow \text{Bot})$
- По функции типа $(A \rightarrow \text{Bot}) \rightarrow \text{Bot}$ нужно получить значение типа A
- Функция типа $(A \rightarrow \text{Bot}) \rightarrow \text{Bot}$ существует, если тип $A \rightarrow \text{Bot}$ населён
- Тип $A \rightarrow \text{Bot}$ населён, если тип A населён

Интуиционистская vs классическая логики

- Выше была интуиционистская логика
- Теорема классической логики: $\neg(\neg A) \rightarrow A$
- Это преобразуется в $((A \rightarrow \text{Bot}) \rightarrow \text{Bot})$
- По функции типа $(A \rightarrow \text{Bot}) \rightarrow \text{Bot}$ нужно получить значение типа A
- Функция типа $(A \rightarrow \text{Bot}) \rightarrow \text{Bot}$ существует, если тип $A \rightarrow \text{Bot}$ населён
- Тип $A \rightarrow \text{Bot}$ населён, если тип A населён

Нужна функция, которая берёт населённый тип и возвращает значение этого типа — это невозможно реализовать в рамках нашей системы

Закон исключённого третьего

$$A \vee \neg A = 1$$

Закон исключённого третьего

$$A \vee \neg A = 1$$

- Закон, верный в классической логике

Закон исключённого третьего

$$A \vee \neg A = 1$$

- Закон, верный в классической логике
- В интуиционистской логике он **неверен!**

Закон исключённого третьего

$$A \vee \neg A = 1$$

- Закон, верный в классической логике
- В интуиционистской логике он **неверен!**
- Подставив вместо A 1 или 0, мы получим верное утверждение

Закон исключённого третьего

$$A \vee \neg A = 1$$

- Закон, верный в классической логике
- В интуиционистской логике он **неверен!**
- Подставив вместо A 1 или 0, мы получим верное утверждение
- Но так делать нельзя

Изоморфизм Карри-Говарда

Системы доказательств и модели вычислений суть **одинаковые** по структуре типы объектов

Неформально

Любой **типизированной** системе соответствует некоторая логика, и наоборот

Неформально

Любой типизированной системе соответствует некоторая логика, и наоборот

Неформально

Любой типизированной системе соответствует некоторая логика, и наоборот

- Зачем мы тогда вводили ограничения? Без них ведь система всё равно оставалась бы типизированной

Неформально

Любой типизированной системе соответствует некоторая логика, и наоборот

- Зачем мы тогда вводили ограничения? Без них ведь система всё равно оставалась бы типизированной
- Логика некоторых систем может оказаться бесполезной или бессмысленной

Запрещённые функции

В Haskell:

- Каждый тип населён значением `undefined` (семантически — forall a. a или \perp)

Запрещённые функции

В Haskell:

- Каждый тип населён значением `undefined` (семантически — forall a . a или \perp)
- Следовательно, можно доказать **любую** теорему

Запрещённые функции

В Haskell:

- Каждый тип населён значением `undefined` (семантически — forall a . a или \perp)
- Следовательно, можно доказать **любую** теорему
- В том числе, можно одновременно доказать A и $\neg A$

Запрещённые функции

В Haskell:

- Каждый тип населён значением `undefined` (семантически — forall a . a или \perp)
- Следовательно, можно доказать **любую** теорему
- В том числе, можно одновременно доказать A и $\neg A$
- Система неконсистентна — в ней истинными являются противоречащие друг другу утверждения

Запрещённые функции

В Haskell:

- Каждый тип населён значением `undefined` (семантически — forall a. a или \perp)
- Следовательно, можно доказать **любую** теорему
- В том числе, можно одновременно доказать A и $\neg A$
- Система неконсистентна — в ней истинными являются противоречащие друг другу утверждения
- Такие логики называются дегенеративными или вырожденными (degenerative)

Запрещённые функции

В Haskell:

- Каждый тип населён значением `undefined` (семантически — forall a . a или \perp)
- Следовательно, можно доказать **любую** теорему
- В том числе, можно одновременно доказать A и $\neg A$
- Система неконсистентна — в ней истинными являются противоречащие друг другу утверждения
- Такие логики называются дегенеративными или вырожденными (degenerative)
- Всё это верно и для функции `error`

Запрет на рекурсию

Пример рекурсивной функции

```
fix f = f (fix f)
fix :: (a -> a) -> a
```

- Населён тип $(a \rightarrow a) \rightarrow a$

Запрет на рекурсию

Пример рекурсивной функции

```
fix f = f (fix f)
fix :: (a -> a) -> a
```

- Населён тип $(a \rightarrow a) \rightarrow a$
- Следовательно, верна теорема $(A \rightarrow A) \rightarrow A$

Запрет на рекурсию

Пример рекурсивной функции

```
fix f = f (fix f)
fix :: (a -> a) -> a
```

- Населён тип $(a \rightarrow a) \rightarrow a$
- Следовательно, верна теорема $(A \rightarrow A) \rightarrow A$
- Её левая часть всегда верна (доказали раньше)

Запрет на рекурсию

Пример рекурсивной функции

```
fix f = f (fix f)
fix :: (a -> a) -> a
```

- Населён тип $(a \rightarrow a) \rightarrow a$
- Следовательно, верна теорема $(A \rightarrow A) \rightarrow A$
- Её левая часть всегда верна (доказали раньше)
- Следовательно, верна правая **для любого A**

Запрет на рекурсию

Пример рекурсивной функции

```
fix f = f (fix f)
fix :: (a -> a) -> a
```

- Населён тип $(a \rightarrow a) \rightarrow a$
- Следовательно, верна теорема $(A \rightarrow A) \rightarrow A$
- Её левая часть всегда верна (доказали раньше)
- Следовательно, верна правая **для любого A**
- Аналогично предыдущему случаю, можем доказать любое утверждение

Запрет на рекурсию в типах

Пример рекурсивного типа

```
data A = Some (A -> A)
```

Запрет на рекурсию в типах

Пример рекурсивного типа

```
data A = Some (A -> A)
```

- В общем и целом, не опасен, просто всегда населён

Запрет на рекурсию в типах

Пример рекурсивного типа

```
data A = Some (A -> A)
```

- В общем и целом, не опасен, просто всегда населён

```
data A = Some (Not A)
```

Запрет на рекурсию в типах

Пример рекурсивного типа

```
data A = Some (A -> A)
```

- В общем и целом, не опасен, просто всегда населён

```
data A = Some (Not A)
```

- Явно вводит в систему логический парадокс: «Утверждение A гласит, что оно неверно»

Запрет на рекурсию в типах

Пример рекурсивного типа

```
data A = Some (A -> A)
```

- В общем и целом, не опасен, просто всегда населён

```
data A = Some (Not A)
```

- Явно вводит в систему логический парадокс: «Утверждение A гласит, что оно неверно»
- Использовать рекурсивные типы можно, но только некоторые

Запрет на рекурсию в типах

Пример рекурсивного типа

```
data A = Some (A -> A)
```

- В общем и целом, не опасен, просто всегда населён

```
data A = Some (Not A)
```

- Явно вводит в систему логический парадокс: «Утверждение A гласит, что оно неверно»
- Использовать рекурсивные типы можно, но только некоторые
- Выделение класса возможных типов — довольно сложная задача

Запрет на рекурсию в типах

Пример рекурсивного типа

```
data A = Some (A -> A)
```

- В общем и целом, не опасен, просто всегда населён

```
data A = Some (Not A)
```

- Явно вводит в систему логический парадокс: «Утверждение A гласит, что оно неверно»
- Использовать рекурсивные типы можно, но только некоторые
- Выделение класса возможных типов — довольно сложная задача
- Эта задача выходит за рамки нашей лекции

Выход из рамок интуиционистской логики

- Требуется введение специальных конструкций по образу и подобию парадигмы Call/CC

Выход из рамок интуиционистской логики

- Требуется введение специальных конструкций по образу и подобию парадигмы Call/CC
- Call with Current Continuation и Continuation Passing Style позволяют доказать закон Пирса и закон двойного отрицания соответственно

Выход из рамок интуиционистской логики

- Требуется введение специальных конструкций по образу и подобию парадигмы Call/CC
- Call with Current Continuation и Continuation Passing Style позволяют доказать закон Пирса и закон двойного отрицания соответственно
 - Закон Пирса: $((A \rightarrow B) \rightarrow A) \rightarrow A$
 - Закон двойного отрицания: $\neg(\neg A) \rightarrow A$

Выход из рамок интуиционистской логики

- Требуется введение специальных конструкций по образу и подобию парадигмы Call/CC
- Call with Current Continuation и Continuation Passing Style позволяют доказать закон Пирса и закон двойного отрицания соответственно
 - Закон Пирса: $((A \rightarrow B) \rightarrow A) \rightarrow A$
 - Закон двойного отрицания: $\neg(\neg A) \rightarrow A$
- Через них можно доказать теорему об исключении третьего

Выход из рамок интуиционистской логики

- Требуется введение специальных конструкций по образу и подобию парадигмы Call/CC
- Call with Current Continuation и Continuation Passing Style позволяют доказать закон Пирса и закон двойного отрицания соответственно
 - Закон Пирса: $((A \rightarrow B) \rightarrow A) \rightarrow A$
 - Закон двойного отрицания: $\neg(\neg A) \rightarrow A$
- Через них можно доказать теорему об исключении третьего
- Существуют языки с поддержкой этих средств, здесь рассматривать не будем

Связь с выводом типов

- Что такое вывод типов?

Связь с выводом типов

- Что такое вывод типов?
 - У нас есть программа, мы ищем её тип

Связь с выводом типов

- Что такое вывод типов?
 - У нас есть программа, мы ищем её тип
- Что такое доказательство теоремы посредством СНС?

Связь с выводом типов

- Что такое вывод типов?
 - У нас есть программа, мы ищем её тип
- Что такое доказательство теоремы посредством СНС?
 - У нас есть тип, мы пытаемся под него придумать программу

Связь с выводом типов

- Что такое вывод типов?
 - У нас есть программа, мы ищем её тип
- Что такое доказательство теоремы посредством СНС?
 - У нас есть тип, мы пытаемся под него придумать программу
- Что такое вывод типов с точки зрения СНС?

Связь с выводом типов

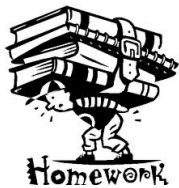
- Что такое вывод типов?
 - У нас есть программа, мы ищем её тип
- Что такое доказательство теоремы посредством СНС?
 - У нас есть тип, мы пытаемся под него придумать программу
- Что такое вывод типов с точки зрения СНС?
 - Полный бред: у нас есть доказательство, мы ищем, что же, собственно, оно доказывает

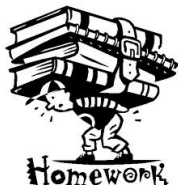
Что мы узнали

- Посмотрели систему доказательств на практике
- Связали логические операции и типы
- Ввели интуиционистскую логику
- Рассмотрели существующие расширения

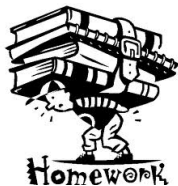
Список литературы

- *Morten Heine B. Sørensen, Pawel Urzyczyn*. Lectures on the Curry-Howard Isomorphism
- *Benjamin C. Pierce*. Types and Programming Languages — ISBN 0-262-16209-1
- Статья «The Curry-Howard Isomorphism» на <http://en.wikibooks.org>

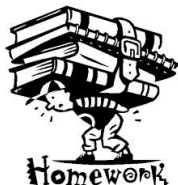




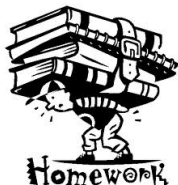
- Возьмите ваш любимый язык программирования



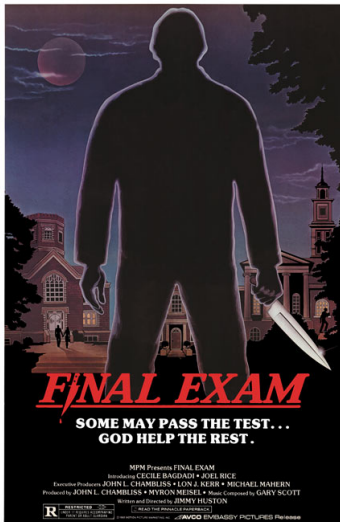
- Возьмите ваш любимый язык программирования
- Рассмотрите его систему типов (или её runtime представление)



- Возьмите ваш любимый язык программирования
- Рассмотрите его систему типов (или её runtime представление)
- Опишите его с точки зрения Curry-Howard Correspondence



- Возьмите ваш любимый язык программирования
- Рассмотрите его систему типов (или её runtime представление)
- Опишите его с точки зрения Curry-Howard Correspondence



FINAL EXAM

**SOME MAY PASS THE TEST...
GOD HELP THE REST.**

MPM Presents FINAL EXAM

Introducing CECILE BAGDADI • JOEL RICE

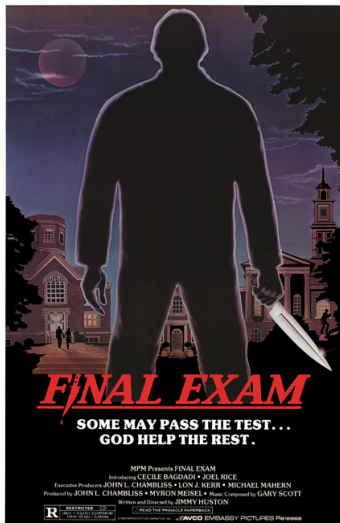
Executive Producers JOHN L. CHAMBLISS • LON J. KERR • MICHAEL MAHERN
Produced by JOHN L. CHAMBLISS • MYRON MEISEL • Music Composed by GARY SCOTT

Written and Directed by JIMMY HUSTON

R RESTRICTED
Under 17 requires accompaniment
of an adult

READ THE PRESS RELEASE HERE

© 1999 MPM. ALL RIGHTS RESERVED. RAVCO EMBASSY PICTURES PRESENTS



See you at the exam! 😊