Изоморфизм Карри-Говарда

Михаил Беляев

30 октября 2018 г.

Истоки

- 1934 г. Хаскелл Карри
  - Типы комбинаторов можно рассматривать как схемы аксиом для импликативной логики
- 1958 г. Хаскелл Карри
  - Система доказательств Гильберта частично совпадает с моделью вычислений логики комбинаторов
- 1969 г. Вильям Алвин Говард
  - Более высокоуровневая система доказательств — «Естественная дедукция» — отображается в типизированное λ-исчисление

Идея

Изоморфизм Карри-Говарда:
Системы доказательств и модели вычислений — суть **одинаковые** по своей структуре типы объектов

Идея

Изоморфизм Карри-Говарда:
Системы доказательств и модели вычислений — суть **одинаковые** по своей структуре типы объектов

Более неформально:
Любой **типизированной** системе соответствует некая логика, и наоборот

Идея

Изоморфизм Карри-Говарда:
Системы доказательств и модели вычислений — суть **одинаковые** по своей структуре типы объектов

Более неформально:
Любой **типизированной** системе соответствует некая логика, и наоборот

Вывод:
Типизированные системы можно использовать для доказательства теорем, и наоборот

Интерпретации

- Доказательство $=$ программа
- Доказываемая формула $=$ тип программы

Интерпретации

- Доказательство $=$ программа
- Доказываемая формула $=$ тип программы

- **Тип** возвращаемого значения функции $\approx$ **теорема** в логике
- **Типы** аргументов $\approx$ гипотезы

Последствия

- Новый класс формальных систем — одновременно и системы доказательств, и типизированные функциональные языки программирования (Coq, Agda, Idris, etc)
- Новые классы типизированных систем, заточенные под доказательства свойств в плюс к программе
- Используется в современной теории типов
- Поднимает новые вопросы в теории вычислений (соответствия различных формальных систем и моделей вычислений)
- Унификация мат. логики и Computer Science (теория категорий пошла ещё дальше, но об этом не сегодня)

Disclaimers

- Всё показанное дальше не является до конца формальным описанием
- Будем всё демонстрировать на примере Haskell, но подойдёт любой язык

Ограничения

Нам нужно только сходящееся подмножество типов, поэтому:

- Только чистые функции (про монады пока забыли)
- Никакой рекурсии в коде
- Никакой рекурсии в типах
  (прости, связный список :-( )
- Никаких `undefined` и `error`
- Только простые `data`-based структуры данных
- Вместо `data A = Ab B | Ac C` будем (где нам удобно) писать просто `B | C`

Доказательство теорем

- Формулируем теорему как некий тип
- Находим значение (любое!), которое подходит под этот тип
- Теорема доказана

Высказывания = типы

- Что означает тип `A -> B`?
- $A \to B$

Высказывания $=$ типы

- Что означает тип `A -> B`?
- $A \to B$

- Если типы `A` и `B` можно интерпретировать логически
- $A \to B$, если `A -> B` — населённый (inhabited) тип

Высказывания = типы

- Что означает тип `A -> B`?
- $A \to B$

- Если типы `A` и `B` можно интерпретировать логически
- $A \to B$, если `A -> B` — населённый (inhabited) тип

Пример:

- `const :: a -> b -> a`
- Соответствует утверждению $\forall a, b. a \to (b \to a)$
  - Дальше будем опускать квантор общности для generic типов и теорем
- Это утверждение верно, потому что функцию с таким типом написать легко: `const x _ = x`

Логические операции

- Оператор импликации $\rightarrow$ соотвествует конструктору функции `->`
- Далее рассмотрим $\wedge$, $\vee$, *true*, *false*, $\neg$
- В классической логике можно выразить все операторы через импликацию (логика, построенная таким образом, называется *импликативной*)

Конъюнкция и дизъюнкция

- $A \wedge B$ истинно если и только если
  - A истинно
  - B истинно
- `(A, B)` можно сконструировать если и только если
  - Можно сконструировать значение типа `A`
  - Можно сконструировать значение типа `B`

Конъюнкция и дизъюнкция

- $A \wedge B$ истинно если и только если
  - A истинно
  - B истинно
- `(A, B)` можно сконструировать если и только если
  - Можно сконструировать значение типа `A`
  - Можно сконструировать значение типа `B`
- Аналогично $A \vee B$ соответствует тип `A | B`

Тип для *true*

- Нужен тип, который всегда населён
- Подойдёт **любой** встроенный
- Для простоты можно взять ( )

Тип для *true*

- Нужен тип, который всегда населён
- Подойдёт **любой** встроенный
- Для простоты можно взять `()`

Следствия:

- `(() | A)` и `(A | ())` населены для любого типа `A`:
  - Т. е. $true \vee A$ и $A \vee true$ всегда истинны
- `((),A)` и `(A,())` ведут себя так же как тип `A`:
  - Соответствуют $true \wedge A$ и $A \wedge true$
- `A -> ()` всегда населён (функцией `unit _ = ()`):
  - $A \to true$ всегда верно

Тип для *false*

- С false всё сложнее
- Нужен заведомо ненаселённый тип (не путать с `()`!)
- В GHC есть «фантомный» тип $\bot$, 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`
- В Scala/Kotlin мы бы взяли тип `Nothing`

Тип для *false*: следствия

- `(Bot, A)` и `(A, Bot)` не населены для любых `A`
  - Т. е. $\mathit{false} \wedge A$ и $A \wedge \mathit{false}$ всегда ложны
- `Bot | A` и `A | Bot` населены только если населён `A`
  - Что соответствует $\mathit{false} \vee A$ и $A \vee \mathit{false}$
- `Bot -> A` населён **всегда** (см. выше функция `cast`)
  - $\mathit{false} \to A$ всегда истинно

Отрицание

Можно ли выразить отрицание через уже определённые операции?

Отрицание

Есть один способ: $\neg A = (A \to \mathit{false})$

```
type Not a = a -> Bot
```

Система доказательств Гильберта

Аксиомы:

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

Правило вывода: Modus Ponens

Система доказательств Гильберта

Аксиомы:

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

Правило вывода: Modus Ponens

Докажем, что $A \to A$

- **Ax2**$[A \,/\, C]$: $(A \to (B \to A)) \to ((A \to B) \to (A \to A))$
- **Ax1**: $(A \to B) \to (A \to A)$
- $[C \to A \,/\, B]$: $(A \to (C \to A)) \to (A \to A)$
- **Ax1**: $A \to A$

Исчисление комбинаторов (SK-исчисление)

Аксиомы:

- ```
  k :: a -> b -> a
  k x _ = x
  ```
  - $A \to (B \to A)$
- ```
  s :: (a -> b -> c) -> (a -> b) -> a -> c
  s x y z = x z $ y z
  ```
  - $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

Правило вывода: Применение функции к аргументу

Докажем, что $A \to A$, т.е. напишем функцию `i :: a -> a`

Исчисление комбинаторов (SK-исчисление)

Аксиомы:

- `k :: a -> b -> a`
  `k x _ = x`
  - $A \to (B \to A)$
- `s :: (a -> b -> c) -> (a -> b) -> a -> c`
  `s x y z = x z $ y z`
  - $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

Правило вывода: Применение функции к аргументу

Докажем, что $A \to A$, т.е. напишем функцию `i :: a -> a`

Доказательство: `i = s k k`

Комбинаторная логика

- 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

Ещё примеры

Каррирование

`(a -> b -> c)` $\equiv$ `((a, b) -> c)`

$(A \to (B \to C)) \equiv (A \wedge B \to C)$

Докажем?

Интуиционистская vs классическая логика

- Всё, что мы рассматривали до этого — это *интуиционистская* логика
- Теорема классической логики $\neg(\neg A) \to A$
- Она недоказуема в системе типов Haskell
- Это преобразуется в `((A -> Bot) -> Bot)`
- По функции типа `(A -> Bot) -> Bot` нужно получить значение типа `A`
- Это невозможно :-(

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

Больше ада!

Re: начало лекции

Изоморфизм Карри-Говарда:
Системы доказательств и модели вычислений — суть одинаковые по своей структуре типы объектов

Более неформально:
Любой **типизированной** системе соответствует некая логика, и наоборот

Нет ли здесь противоречия? Зачем мы тогда вводили ограничения?

Больше ада!

- Да, **любой** типизированной системе соответствует некая логика
- Вот только не все логики одинаково полезны
- Бывают логики бесполезные и/или бессмысленные

За что мы запретили `undefined`?

В Haskell:

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

За что мы запретили рекурсию в функциях?

```
a x = a x
```

Имеет любой тип и может доказать любую теорему

За что мы запретили рекурсию в типах?

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

Этот тип не особо опасен, просто всегда населён

За что мы запретили рекурсию в типах?

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

Этот тип не особо опасен, просто всегда населён

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

«Утверждение A гласит, что оно неверно», упс

- 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

- Требует введения 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: $((A \to B) \to A) \to A$
  - Закон двойного отрицания: $\neg(\neg A) \to A$
- Через них можно доказать теорему об исключении третьего
- Существуют языки со встроенной поддержкой этих средств, здесь рассматривать не будем
  - Например, Scheme и Racket

ИКГ vs вывод типов

- Доказательство теоремы с помощью типов и вывод типов из программы — это **дуальные** задачи
- С точки зрения друг друга — полный бред
  - «У нас есть тип, нужно придумать под него программу»
  - «У нас есть доказательство, нужно придумать, что же оно доказывает»

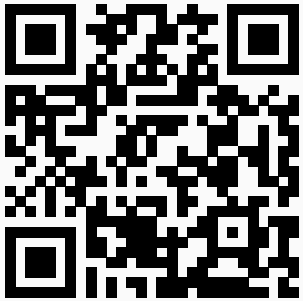
Расширения

Изоморфизм Карри-Говарда-Ламбека

- Те же ребята + закрытые декартовы категории
- Мы рассматривать здесь не будем

Сегодня мы многое поняли

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



http://kspt.icc.spbstu.ru/course/lang

belyaev@kspt.icc.spbstu.ru

КСПТ | КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ СПбПУ

ПОЛИТЕХ
Санкт-Петербургский политехнический университет Петра Великого