Доказательства методом бесконечного спуска и их формализация на Агде

С.А.Романенко

23 мая 2017

Типы - это не совсем множества

> $x : A$ означает, что "x имеет тип A", или что "x - это A".

- В конструктивной математике, "типы" похожи на биологические виды.
- Когда биолог видит собаку, он может распознать, что это - собака.
- Но никто не знает, сколько есть собак на Земле, и не может их ни предъявить, ни перечислить.

Что такое "подмножество"?

- В теории множеств, `A ⊆ B` означает
  `∀ x → x ∈ A → x ∈ B`.
  Например: `Собаки ⊆ Животные`.
- А если **нет** понятия x ∈ A? "Подмножество" - это что? 😩
- Взамен, можно работать с **предикатами** и **импликацией**:
  `∀ (x : Живность) → Собака x → Животное x`
  `∀ (n : ℕ) → Even (n + n)`

А каков тип у предикатов (и у типов)?

- `ℕ : Set`
- `Even : ℕ → Set`
- `Set` - сокращение для `Set₀`, `Set₀ : Set₁`, `Set₁ : Set₂`, ...

Числа Пеано

- **zero:** `zero : ℕ`.
- **suc:** `n : ℕ → suc n : ℕ`.
- **suc-inj:** `suc m = suc n → m = n`.
- **case$\mathbb{N}$:** `n : ℕ → (n = zero)` либо `(∃ m → n = suc m)`.
- **ind$\mathbb{N}$:** `P zero → (∀ m → P m → P (suc m)) → (∀ n → P n)`.

На Агде всё вышесказанное выражается так:

```
data ℕ : Set where
  zero : ℕ
  suc  : (n : ℕ) → ℕ
```

> **suc-inj**, **case$\mathbb{N}$** и **ind$\mathbb{N}$** подразумеваются Агдой
> (и могут быть доказаны на Агде в явном виде).

Пример: определение сложения

- `zero + m = m`
- `(suc n) + m = suc (n + m)`

То же самое - на Агде:

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
suc n + m = suc (n + m)
```

Тонкости:

- В Агде определения асимметричны
  (задача слева сводится к задаче справа).
- Агда проверяет, что вычисление завершается
  (задача сводится к более простой).

Докажем на Агде `suc-inj` и `caseℕ`

```
suc-inj : ∀ {m n} → suc m ≡ suc n → m ≡ n
suc-inj {m} {.m} refl = refl {x = m}
```

Тавтологическое доказательство!

`refl : ∀ {x} → x ≡ x`. Если дано `suc k ≡ suc k`, то `k ≡ k`.

```
caseℕ : ∀ n → (n ≡ zero) ⊎ (∃ λ k → n ≡ suc k)
caseℕ zero    = inj₁ refl
caseℕ (suc k) = inj₂ (k , refl)
```

Разбираем два случая, сопоставляя аргумент с образцом.

Навешиваем `inj₁` или `inj₂`, чтобы пометить случаи.

Доказательство существования такого `x`, что `P x`, выдается в виде пары `(x , px)`, где `px : P x`.

Пример: докажем "вручную", что `n + 0 = n`

> Будем, для краткости, писать `0` вместо `zero`.
> (Агда это позволяет.)

Теорема: `n + 0 = n`.

Доказательство ("по индукции").

- Пусть `n = 0`. Тогда `0 + 0 = 0` *{по определению +}*.
- Пусть `n = suc k`. Предполагая, что `k + 0 = k`, докажем `(suc k) + 0 = suc k`.
  - `(suc k) + 0`
    = *{по определению +}*
  - `suc (k + 0)`
  - = *{конгруэнтность* `suc`*, гипотеза индукции* `k + 0 = k`*}*
  - `suc k`

Пример: докажем на Агде, что `n + 0 = n`

```
+-zero : ∀ n → n + zero ≡ n

+-zero zero =
  zero + zero ≡⟨⟩ zero ∎
+-zero (suc k) =
  (suc k) + zero
    ≡⟨⟩
  suc (k + zero)
    ≡⟨ cong suc (+-zero k) ⟩
  suc k ∎
```

"Соответствие Карри-Ховарда":

- Утверждение (`n + zero ≡ n`) ~ тип.
- Доказательство (`+-zero n`) ~ "жилец" этого типа .
- Импликация (`→`) ~ функция.
- Квантор общности (`∀`) ~ параметр функции.

Математическая индукция (Mathematical induction)

Augustus de Morgan (**1838**)

На Агде доказывается в качестве теоремы:

```
indℕ : {P : ℕ → Set}
  (base : P zero) (step : ∀ m → P m → P (suc m))
  (n : ℕ) → P n
indℕ base step zero =
  p0
indℕ base step (suc k) =
  step k (indℕ base step k)
```

`indℕ` - это, по сути, "примитивная рекурсия"!

```
primℕ : (base : ℕ) (step : ℕ → ℕ → ℕ) (n : ℕ) → ℕ
primℕ base step zero = base
primℕ base step (suc n) =
  step n (primℕ base step n)
```

Что лучше: `primℕ` / `indℕ` или прямая рекурсия?

За `primℕ` / `indℕ`.

- Соответствует структурному программированию (`while` вместо `goto`).
- Проще выполнять ***некоторые*** преобразования над программами/доказательствами.

Против `primℕ/indℕ`.

- При построении программы/доказательства не всегда заранее ясна его структура.
- Не всегда можно уложиться в `primℕ/indℕ`.

```
ack : ℕ → ℕ → ℕ
ack 0 n = 1
ack (suc m) 0 = ack m 1
ack (suc m) (suc n) = ack m (ack (suc m) n)
```

Пример: корректность компилятора

Арифметические выражения.

```
data Tm : Set where
  val : (n : ℕ) → Tm
  _⊕_  : (t1 t2 : Tm) → Tm
```

Интерпретатор

```
eval : Tm → ℕ
eval (val n) = n
eval (t1 ⊕ t2) = eval t1 + eval t2
```

Исполняемый код

Пусть `c : Code i j`. Тогда исполнение `c` применительно к стеку `s` глубины `i`, порождает стек глубины `j`.

```
data Code : (i j : ℕ) → Set where
  seq  : ∀ {i j k}
           (c1 : Code i j) (c2 : Code j k) → Code i k
  push : ∀ {i} (n : ℕ) → Code i (1 + i)
  add  : ∀ {i} → Code (2 + i) (1 + i)
```

Компилятор

```
compile : ∀ {i} (t : Tm) → Code i (1 + i)
compile (val n) = push n
compile (t1 ⊕ t2) =
  seq (seq (compile t1) (compile t2)) add
```

Интерпретатор кода

`s : Stack i` означает, что `s` - это стек глубины `i`, представленный вектором длины `i`.

Пустой стек - это `[]`, а непустой - `n :: s`, где `n` - число на вершине стека, а `s` - остаток стека.

```
Stack : ℕ → Set
Stack i = Vec ℕ i

exec : ∀ {i j} (c : Code i j) (s : Stack i) → Stack j
exec (seq c1 c2) s = exec c2 (exec c1 s)
exec (push n) s = n :: s
exec add (n2 :: n1 :: s) = (n1 + n2) :: s
```

Корректность `compile` и `exec` по отношению к `eval`

```
correct : ∀ {i} (t : Tm) (s : Stack i) →
  exec {i} (compile t) s ≡ eval t :: s
correct (val n) s = refl
correct (t1 ⊕ t2) s =
  exec (compile (t1 ⊕ t2)) s
    ≡⟨⟩
  exec (seq (seq c1 c2) add) s
    ≡⟨⟩
  exec add (exec c2 (exec c1 s))
    ≡⟨ cong (exec add ∘ exec c2) (correct t1 s) ⟩
  exec add (exec c2 (n1 :: s))
    ≡⟨ cong (exec add) (correct t2 (n1 :: s)) ⟩
  exec add (n2 :: n1 :: s)
    ≡⟨⟩
  n1 + n2 :: s
    ≡⟨⟩
  eval (t1 ⊕ t2) :: s ∎
  where
  n1 = eval t1; n2 = eval t2;
  c1 = compile t1; c2 = compile t2
```

Бесконечный спуск (infinite descent)

Pierre de Fermat (**1659**)

На Агде доказывается в качестве теоремы:

```
descℕ : {P : ℕ → Set}
  (¬p0 : P zero → ⊥) (down : ∀ m → P (suc m) → P m)
  (n : ℕ) → P n → ⊥
descℕ ¬p0 down zero p0 =
  ¬p0 p0
descℕ ¬p0 down (suc k) p$1+k =
  descℕ ¬p0 down k (down k p$1+k)
```

> Напоминание: `¬ P n` - это сокращение для `P n → ⊥`.
> "Предположив `P n`, приходим к противоречию".

Что чем удобнее доказывать?

Позитивные факты - индукцией!

- $n + 0 = n$

Негативные факты - спуском!

- $n \neq 1 + n$
- $\sqrt{2}$ - иррационально, ибо не существует таких m и n, что $n \neq 0$ и $m^2 = 2 * n^2$.

Пример: докажем, что `n ≡ suc n → ⊥`

Предположим, что дано такое `n`, что `n = suc n`.

- Пусть `n = 0`. Тогда дано `0 = suc 0`.
  Что не может быть дано, ибо противоречит **case**ℕ!
- Пусть `n = suc k`. Тогда дано `suc k = suc (suc k)`.
  - `suc k = suc (suc k)` ⟶ **{suc-inj}** `k = suc k`.
  - `k < n`. Доказываем (рекурсивно), что `k ≠ suc k`.
  - Из `k ≠ suc k` и `k = suc k` следует `⊥`.

То же самое доказательство на Агде:

```
n≢1+n : ∀ n → n ≡ suc n → ⊥
n≢1+n zero ()
n≢1+n (suc k) 1+k≡2+k =
  n≢1+n k (suc-inj 1+k≡2+k)
```

Особенности доказательств спуском на Агде

- "Технически", доказательства методом бесконечного спуска - это тоже доказательства по индукции (в виде рекурсивных функций).
- При доказательстве негативного утверждения `¬ P`, это утверждение переводится в позитивную форму `P` и втаскивается в аргументы функции. Это, неформально, можно истолковывать как "предположим, что `P`.

Агитация за бесконечный спуск

- Claus-Peter Wirth; **Computer-assisted human-oriented inductive theorem proving by descente infinie—a manifesto.** Log J IGPL 2012; 20 (6): 1046-1063. https://doi.org/10.1093/jigpal/jzr048

> From the ancient Greeks until today, mathematical theories, notions and proofs are not developed the way they are documented. This difference does not only consist of the iterative deepening of the development and the omission of easily reconstructible parts; also the global order of presentation in publication more often than not differs from the order of development. This results in the famous eureka steps, which puzzle the freshmen in mathematics.

The Cyclist Framework and Provers

Cyclist is two things:

- A generic, cyclic theorem prover framework. It can be used for constructing theorem provers that employ cyclic proof (usually, in order to treat inductively defined predicates) for practically any logic.
- A collection of theorem provers and other tools, with a focus on Separation Logic with inductive definitions, hereafter abbreviated as SL.

http://www.cyclist-prover.org/
https://github.com/ngorogiannis/cyclist

Как всё это относится к Агде?

- Агда позволяет изображать индукцию в виде рекурсивных функций. Нет надобности втискиваться в ограниченное число схем индукции.
- Агда, в основном, рассчитана изготовление доказательств "вручную", в то время, как Cyclist (и ему подобные) пытаются строить доказательства автоматически.

Посмотрим на примеры индукции, которая плохо втискивается в фиксированные схемы.

Взаимное определение двух типов данных

```
mutual

  data Even : ℕ → Set where
    even0  : Even zero
    even1 : ∀ {n} → Odd n → Even (suc n)

  data Odd : ℕ → Set where
    odd1 : ∀ {n} → Even n → Odd (suc n)
```

Например:

```
odd-1 : Odd 1
odd-1 = odd1 even0

even-2 : Even 2
even-2 = even1 (odd1 even0)
```

Инверсия `Even` и `Odd`

```
¬odd-0 : ¬ Odd zero
¬odd-0 ()

even-suc : ∀ {n} → Even (suc n) → Odd n
even-suc (even1 odd-n) = odd-n

odd-suc : ∀ {n} → Odd (suc n) → Even n
odd-suc (odd1 even-n) = even-n
```

> Инверсия полезна для доказательств методом спуска!

`Even (m + m)`: "традиционная" индукция

```
even-2* : ∀ n → Even (n + n)
even-2* zero = even0
even-2* (suc k) = step (even-2* k)
  where
  open Related.EquationalReasoning renaming (sym to ~sym)
  step : Even (k + k) → Even (suc k + suc k)
  step =
    Even (k + k)
      ~⟨ even1 ∘ odd1 ⟩
    Even (suc (suc (k + k)))
      ≡⟨ cong (Even ∘ suc) (sym $ +-suc k k) ⟩
    Even (suc (k + suc k))
      ≡⟨ refl ⟩
    Even (suc k + suc k)
    ∎
```

Бесконечный спуск в духе Ферма

```
¬odd-2* : ∀ n → Odd (n + n) → ⊥
¬odd-2* zero odd-0 =
  ¬odd-0 odd-0
¬odd-2* (suc k) h =
  ¬odd-2* k (down h)
  where
  open Related.EquationalReasoning renaming (sym to ∼sym)
  down : Odd (suc k + suc k) → Odd (k + k)
  down =
    Odd (suc k + suc k)
      ≡⟨ refl ⟩
    Odd (suc (k + suc k))
      ≡⟨ cong (Odd ∘ suc) (+-suc k k) ⟩
    Odd (suc (suc (k + k)))
      ∼⟨ even-suc ∘ odd-suc ⟩
    Odd (k + k)
    ∎
```

Исходные тексты примеров

Большое количество "учебных" примеров - в репозитории

- https://github.com/sergei-romanenko/agda-samples

В том числе, по поводу доказательств по индукции:

- `06-Induction.agda` и `06-InductionSol.agda`.
- `08-WellFounded.agda`.
- `14-InfiniteDescent`.agda.
- `SmallStep/SmallStepDepRec.agda`.

Также, в репозитории

- https://github.com/sergei-romanenko/agda-Pythagoras

находится доказательство того, что $\sqrt{2}$ - иррационально.

Выводы

- Агда позволяет единообразно записывать доказательства по индукции и методом спуска в виде рекурсивных функций.
- Доказательства легче придумывать не втискивая их в ограниченный набор схем индукции.
- При автоматическом поиске доказательств - тоже легче их строить в "свободной форме".