ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № 3(21), 2014, с. 27–50

УДК 510.252, 004.432.42

С. Д. Мешвелиани

О зависимых типах и интуиционизме в программировании математики

Аннотация. Обсуждается практическая возможность доказуемого программирования математики на основе подхода конструктивизма и применения языка с зависимыми типами (dependent types, proof carrying code). Принципы конструктивизма и доказуемого программирования объясняются на примерах. Рассмотрение опирается на опыт реализации на языке Agda небольшой библиотеки вычислительной алгебры, включающей арифметику области остатков $R/(b)$ для евклидова кольца R.

Ключевые слова и фразы: конструктивная математика, алгебра, зависимые типы, Coq, Agda, Haskell.

Введение

Данная статья описывает итог исследования по подбору средства (языка) для наиболее адекватного программирования математических вычислений. Она опирается на опыт программирования библиотеки алгоритмов символьных вычислений в алгебре.

Разработчики универсальных языков программирования преследовали каждый свою цель, вовсе не обязательно — представление алгебраических областей. Целью же статьи является найти универсальный язык программирования, обладающий возможно большей математической выразительностью, и объяснить его достоинства в сравнении с другими языками.

История этого поиска и опыта программирования состоит из крупных шагов, соответствующих трём классам языков:

(1) языки обобщённого программирования (generic programming),
(2) чисто функциональные языки,

Работа поддержана Программой № 16 Фундаментальных исследований Президиума Российской академии наук («Фундаментальные проблемы системного программирования»).

(3) языки с зависимыми типами (dependent types).

Ниже объясняется, почему нужен класс языков (1). Представителем классов (1) и (2) является, например, язык Haskell [1]. На нём автор статьи составил в 1990-х годах библиотеку DoCon [2] для методов коммутативной алгебры. Понятие чисто функционального программирования и свойства языка Haskell здесь предполагаются известными читателю.

Наконец, рассматриваем язык Agda [3], принадлежащий классам (1), (2), (3) и лишь недавно получивший относительно работоспособную реализацию. Настоящая статья в основном посвящена объяснению, почему этот язык больше подходит для описания алгебраических областей и задания вычислений в математике.

О похожих работах: данная статья есть исправленное и развёрнутое изложение работ [4] (первый опыт), [5] (тезисы).

Об источниках по математике: математические понятия, используемые в этой статье, можно найти в [6] — алгебра, [7, 8] — вычислительная алгебра.

В дальнейшем речь пойдёт о способах программирования математических вычислений и доказательств на языке Agda [3].

0.1. Обобщённое программирование

Часто программа для математического вычисления действует в различных областях единообразным способом. Например, нахождение наибольшего общего делителя НОД двух элементов в учебниках алгебры описано в виде алгоритма вычисления в любом *евклидовом кольце* E. В качестве E можно подставить бесконечно много разных областей. Примеры: целые числа $\mathbb{Z}$, область $\mathbb{Q}[x]$ многочленов с рациональными коэффициентами, область $(\mathbb{Z}/(p))[x]$ многочленов с коэффициентами — целыми числами по модулю простого p, область $\mathbb{Q}(t)[x]$ многочленов с коэффициентами, являющимися рациональными функциями над рациональными числами.

Этот подход давно нашёл отражение в программировании: появились языки программирования с абстрактными/полиморфными типами (generic programming), в которых, например, программа для наибольшего общего делителя запишется единожды для всех областей соответствующего класса.

При этом каждый класс математических областей (класс всех групп, класс всех полей, и так далее) представлен в языке

программирования *классом значений* (например, «data class» в языке Haskell), а каждая конкретная область из данного класса представлена *типом* и набором *случаев классов* (например, «class instance» в языке Haskell) для этого типа.

Например, класс всех групп по сложению в языке Haskell [1] может быть записан в виде объявления

```
class Group a where (+) :: a -> a -> a
                    0   :: a
                    neg :: a -> a
```

(это суть только сигнатуры действий, сами законы группы в Haskell -программе не могут быть объявлены). Далее, можно для области `(a, b)` пар задать случай группы — при условии, что `a` и `b` являются группами:

```
instance (Group a, Group b) => Group (a, b)
                               where
                               (x, y) + (x', y') = (x + y, x' + y')
                               0                 = (0, 0)
                               neg (x, y)        = (neg x, neg y)
```

Это есть программа для функтора, строящего группу прямого произведения из любой пары групп.

В других языках обобщённого программирования возможна другая терминология, но основа подхода (применительно к нашему предмету) есть примерно та же.

Данный подход воплощён на практике в обширной вычислительной библиотеке Axiom [9] для математики, использующей язык Spad. Разработан также язык Aldor [10], который есть развитие языка Spad.

Библиотека вычислительной алгебры DoCon [2] написана автором статьи на языке Haskell. Главными отличиями языка Haskell от языка Aldor являются чистая функциональность и «ленивый» способ вычисления. Нежелательное отличие языка Haskell — отсутствие *зависимых типов* — обсудим ниже.

Дополнительной причиной разработки библиотеки DoCon в 1990-х было коммерческое (на то время) положение системы Axiom.

0.2. Проблема области, зависящей от параметра

Рассмотрим пример: область $D = \mathbb{Z}/(m)$ остатков целых чисел по модулю m зависит от параметра m. Известны различные

методы вычисления, в которых определённые равенства в области D проверяются для множества значений m, которое велико, и даже становится известным не раньше, чем во время выполнения программы. Кроме того, законность применения того или иного известного метода вычисления к области D зависит от значения m. Например, некоторые методы требуют, чтобы m было простым, иначе результат может быть неправильным.

Есть много подобных примеров.

Такая область не может быть задана в языке Haskell как *тип* и набор случаев классов для него. Ибо типы в этом языке распознаются статически (во время компиляции). То же относится, например, и к языку ML.

Поэтому в программах библиотеки DoCon применяется явная символьная кодировка области — в дополнение ко множеству случаев классов, связанных с типом. Это нежелательное усложнение системы программирования есть плата за математическую выразительность. Здесь, как и в некоторых других системах, применяется *интерпретатор* для вычисления символьного выражения области.

Подход интерпретатора к проверке соответствия области обладает очевидным недостатком против статической проверки. Например, программа может прерваться по ошибке несоответствия области лишь после нескольких часов вычисления.

0.3. Зависимые типы и цель исследования

С некоторого времени изобретены и реализованы языки программирования, в которых полностью решена вышеописанная проблема динамического параметра: Aldor [10], система Coq [11], Agda [12], [3].

Это суть языки с *зависимыми типами*. В них тип может зависеть от обычных значений, и вычисления на типах применяются так же, как вычисления на обычных значениях.

Представляется, что до 2005 года ни одна из этих реализаций (возможно, кроме Coq) не была работоспособной на практике. К 2012-му году эти разработки усилились и стали по-настоящему применимы.

С 2013 года автор данной статьи начал разрабатывать разновидность DoCon-A библиотеки DoCon, основанную на применении языка Agda. Язык Agda выбран из-за (1) его чистой функциональности и

«ленивого» способа вычисления и (2) близости к языку Haskell, на котором нами написаны наши библиотека и доказыватель[1] Dumatel.

Целью исследования и работы является:

- переписать библиотеку DoCon на язык Agda как на более подходящий язык (будем также надеяться на усиление самих реализации и библиотеки системы Agda),
- исследовать на этом содержательном примере возможности конструктивной математики и доказуемого программирования на зависимых типах.

В дальнейшем в этой статье язык Agda используется как программная модель конструктивизма и системы зависимых типов.

Задание алгоритмов посредством вычислений с зависимыми типами основано на подходе конструктивной математики [**13**] (объяснения даны ниже).

Теоретической основой вычислений с зависимыми типами является конструктивная теория типов Мартина Лёфа [**14**].

В дальнейшем изложении более подробно объясняется подход конструктивизма и зависимых типов, рассматриваются примеры, обсуждаются черты библиотеки вычислительной алгебры DoCon-A.

1. Некторые черты языка Agda и простейшие примеры программ с завимыми типами

1.1. Математические символы в идентификаторах

Надо объяснить некоторые типографские и лексические черты языка Agda, ибо без этого большинство Agda программ понять невозможно.

Agda воспринимает символы в кодировке UTF-8. Это даёт возможность ставить в исходную программу математические обозначения. Например: $\mathbb{N}$, $\approx$, $\not\approx$, $\not\equiv$, $\leq$, $\not\leq$, $\circ$, $\bullet$, x^{-1}, $\neg$.

UTF символы вводятся в программу через текстовый редактор emacs, настроенный на режим agda-mode. В этом режиме emacs также правильно отображает эти символы на экран.

Имена (идентификаторы) в Agda - программе разделяются (в основном) только пробелом. Так что оператор или переменная отделяется пробелами.

Пример: рассмотрим отрывок кода

[1] «Доказыватель» это перевод на русский язык слова «prover»

```
m : ℕ
m = foo1
n = foo2

2*n≥m : 2 * n ≥ m   -- объявление принадлежности типу
2*n≥m = f m n       -- применение функции  f

p = g 2*n≥m    -- применение функции g к значению из типа (2 * n ≥ m)
```

В нём `2*n≥m` есть имя переменной. Оно обозначает значение в типе `2 * n ≥ m`. А символы ‘`*`’, ‘`≥`’ в выражении для этого типа обозначают соответственно оператор и конструктор типа — ибо они отделены пробелами.

Заметим также, что имя `2*n≥m` переменной является «говорящим»: в нём закодирован (для читателя) тип этой переменной, то есть смысл её значения.

Итак: лексические правила языка Agda дают: а) математические символы, б) больше возможности для мнемоники в именах.

1.2. Пример: построение рационального числа

Пусть в программе функция `f : ℕ → ℕ` задаёт отображение натуральных чисел, а функция

```
g : ℕ → Rational
g n =  record{ num = 1; denom = f n; denom≠0 = f<n>≠0 }
       where
       f<n>≠0 =  <доказательство>
```

использует `f` для получения рационального числа `1/(f n)`.

Дадим пояснения:

‘`:`’ означает «принадлежит типу»,
‘`T → U`’ означает тип всех функций из типа `T` в тип `U`.

На языке Agda тип рациональных чисел можно объявить как *запись*

```
record Rational : Set where
                      field num      : ℕ
                            denom    : ℕ
                            denom/=0 : ¬ (denom ≡ 0)
```

В типе данных `Rational` поля `num` и `denom` суть поля числителя и знаменателя, `denom≠0` есть поле записи для свидетельства (доказательства) того, что `denom` не равен нулю (иначе дробь `num/denom` не определена) (условие несократимости здесь пропускаем).

Выражение `(denom ≡ 0)` есть *семейство типов* `T`, зависящее от параметра (*индекса*) `denom`. Любое данное `d : T` этого типа, считается доказательством утверждения «`denom = 0`».

Так что строка `g n = ...` строит рациональное число.

В функции `g` часть `<доказательство>` есть алгоритм построения элемента `ne` типа `nT = ¬ (f n ≡ 0)` (то есть — доказательства неравенства нулю). Это формальное доказательство использует определение равенства `_≡_`, определение функции `f` (которое мы опустили), а алгоритм построения свидетельства (который мы опустили) соответствует доказательству неравенства `f n ≠ 0` для любого `n`. Это доказательство может быть, например, проведено индукцией по `n`.

Проверяльщик типов (type checker) строит это доказательство по исходному коду функции, строящей значение в неком данном типе `T`, зависящем от параметра `n`. Проверка соответствия типов делается проверяльщиком путём вычисления на выражениях параметрических типов.

Что важно: эта проверка делается до исполнения программы, она не требует задания числового значения для `n`. Всё это выражает построение на этапе проверки типов доказательства соответствующего утверждения с квантором всеобщности для переменной `n`.

1.3. Конструктивные логические связки

В доказуемом программировании используется приём выражения конструктивной *импликации* через отображение областей свидетельств (доказательств): если типы `T` и `U` соответствуют утверждениям `P` и `Q`, то тип $\texttt{T} \to \texttt{U}$ выражает импликацию `P => Q`. Ибо любая функция `f` из типа $\texttt{T} \to \texttt{U}$ отображает любое свидетельство `t : T` для `P` в свидетельство `(f t) : U` для `Q`.

Конструктивная конъюнкция `P` и `Q` выражается прямым произведением типов: $\texttt{T} \times \texttt{U}$; элемент произведения записывается как пара `(t , u)`.

Конструктивная дизъюнкция `P` и `Q` выражается разделённой суммой типов: $\texttt{T} \uplus \texttt{U}$; элемент суммы записывается как $\texttt{inj}_1\ \texttt{t}$ или $\texttt{inj}_2\ \texttt{u}$, где inj_1 и inj_2 суть конструкторы вложения в сумму из `T` и `U` соответственно.

Следующий пример даёт более определённое представление о доказуемом программировании на зависимых типах.

1.4. Пример: определение полугруппы

Оно может быть записано в виде

```
record Semigroup (A : Setoid) : Set
  where
  open Setoid A using (_≈_) renaming (Carrier to C)
  field
    _•_   : C → C → C
    cong• : (x y x' y' : C) → x ≈ x' → y ≈ y' → (x • y) ≈ (x' • y')
    assoc : (x y z : C) → (x • y) • z ≈ x • (y • z)
```

Поясним. Сетоид (`Setoid`) есть множество `Carrier` с определённым на ним отношением эквивалентности `_≈_`. Это определение взято из стандартной библиотеки. Отношение `_≈_` программируется пользователем для каждого примера сетоида.

Полугруппа есть сетоид, на носителе `C` которого определено действие `_•_`, удовлетворяющее законам `cong•`, `assoc`.

Поле `cong•` задаёт закон конгруэнтности — согласованности отношения `_≈_` и действия `_•_`. Это выражено в виде типа функции. Всякая функция этого типа (это должен быть алгоритм, сопровождаемый доказательством завершаемости) есть доказательство этого закона для соответствующей полугруппы. В выражение типа входят индексы `x y x' y'`. Функция `cong•` отображает любую такую четвёрку и доказательства для `x ≈ x'`, `y ≈ y'` в доказательство равенства `(x • y) ≈ (x' • y')`. Сигнатура `cong•` есть конструктивная запись утверждения: *для любых* `x, y, x', y'` *из* `C` *(если* `x ≈ x'` *и* `y ≈ y'`*, то* `(x • y) ≈ (x' • y')`*)*.

Видно, что типы, из которых составлена сигнатура для `cong•`, *зависят от значений* `x, y, x', y'`. И это позволяет выразить в виде семейства типов утверждение о действии `_•_` и отношении `_≈_`.

Также здесь используется конструктивное представление импликации (в виде типа, через конструктор `_→_`) — как это описано в предыдущем подразделе.

`assoc` имеет тип свидетельства ассоциативности для `_•_`.

1.5. Пример: задание полугруппы натуральных чисел

Выше дано программное определение полугруппы вообще. А случай полугруппы натуральных чисел по сложению задаётся такой программой:

```
nat+semigroup : Semigroup Nat.setoid
nat+semigroup =
                record{ _•_ = _+_; cong• = cong+; assoc = assoc+ }
```

```
where
_+_ : ℕ → ℕ → ℕ    -- сложение в унарной системе
0       + n = n
(suc m) + n = suc (m + n)

_=n_ =  Setoid._≈_ Nat.setoid      -- равенство натуральных чисел

assoc+ : (x y z : ℕ) → (x + y) + z =n x + (y + z)
assoc+ 0       y z =  refl
assoc+ (suc x) y z =
            begin
              ((suc x) + y) + z    =n[ refl ]
              suc ((x + y) + z)    =n[ PE.cong suc (assoc+ x y z) ]
              suc (x + (y + z))    =n[ refl ]
              (suc x) + (y + z)
            □

cong• =  < пропускаем >
```

Дадим пояснения.

Натуральные числа типа `ℕ` записываются в унарной системе, через конструктор `suc` — «следующий».

`Nat.setoid` есть сетоид натуральных чисел, взятый из стандартной библиотеки.

В качестве полугрупповой операции в полугруппу подставляется `_+_`. Две строки после сигнатуры для `_+_` задают алгоритм вычисления этой операции. Agda сама распознаёт его завершаемость, так как во втором предложении функция `_+_` вызывается справа на синтаксически меньшей паре аргументов, чем слева.

Функция `assoc+` есть доказательство ассоциативности для операции `_+_`. Доказательство сделано индукцией по построению первого аргумента. В первом предложении конструктор `refl` значит, что равенство `(0 + y) + z =n 0 + (y + z)` доказывается вычислением выражений левой и правой частей по определению функции `_+_` (это есть сведение выражения типа) и последующим применением закона рефлексивности (`X =n X`).

Во втором предложении правая часть есть доказательство равенства `((suc x) + y) + z =n (suc x) + (y + z)`. Оно представлено тремя последовательными тождественными преобразованиями, в каждой строке справа выражение `=n[ ...]` показывает композицию правил, по которым из выражения в текущей строке слева получается равное ему выражение в следующей строке.

Так, код `=n[ PE.cong suc (assoc+ x y z) ]` означает, что применяется закон ассоциативности к подвыражению `(x + y) + z`, а потом применяется закон конгруэнтности `suc` по отношению к `_=n_`.

1.5.1. *Отступление: о конгруэнтности функций относительно равенства*

Заметим: в алгебре мы всегда имеем дело с *теорией с равенством* (если `x ≈ y`, то `f x ≈ f y`, `x + a ≈ y + a`, ...). В учебниках алгебры эта конгруэнтность операций учитывается неявно, как умалчиваемая очевидность.

Но в языке Agda нужно явно указывать, где конгруэнтность имеет место, и явно использовать её в доказательствах (как это сделано выше в выражении `PE.cong suc (...)`), иначе проверяльщик типов не пропустит доказательства. Это естественно, ибо в произвольной программе итог не обязан в общем случае быть согласован с тем или иным отношением эквивалентности на аргументах, тем более, что последнее может задаваться программистом.

Вернёмся к последнему доказательству в примере.

В конструкции

```
(begin ...=n[ ...] ...  □)
```

`begin_` есть на самом деле префиксная функция, `_ □` есть постфиксная функция, `_=n[_]_` есть инфиксная функция трёх аргументов (её имя получено переименованием (`'renaming'`) при импорте некоторого библиотечного оператора, этот импорт мы в тексте программы пропустили).

И вот, все эти функции запрограммированы на языке Agda и содержатся в стандартной библиотеке. Вся эта конструкция есть композиция вызовов библиотечных функций, которая имеет вид *новой конструкции языка*. Используя этот изящный приём, программист и сам может по сему образцу по сути дела расширять язык.

Это есть замечательное свойство языка Agda:

доказательства пишутся на том же языке, что и программы.

Согласно текущему опыту представляется, что особый язык для доказательств не нужен. Развитие библиотеки DoCon-A покажет, оправдана ли эта надежда.

2. Библиотека DoCon-A как реализация части конструктивной математики

Конструктивная математика [13] предполагает, что всякий объект должен строиться по предъявленному алгоритму, и к алгоритму должно быть приложено доказательство его завершаемости на каждом данном.

Например, библиотека DoCon содержит алгоритмы для решения линейных систем, нахождения базиса Грёбнера ([7], Дополнения, I), действий с дробями над различными областями, разложения многочлена на неприводимые над некоторыми областями, и многие другие. Для этих алгоритмов, как и для огромного множества других известных методов, действительно, вполне осуществимо на практике формальное доказательство завершаемости.

2.1. Проблемы поиска конструктивных доказательств

(1) В большом количестве лёгких случаев проверяльщик типов сам находит доказательство завершаемости, исходя из обозрения рекурсивного вызова функции на структурно синтаксически меньшем аргументе.

(2) В других случаях в программу (функцию) можно добавить аргумент — счётчик и записать формальное доказательство завершаемости, сравнивая значение счётчика с «размерами» аргументов. Это часто делается в учебниках, но здесь доказательство является формальным и включено в программу.[2]

(3) Но в некоторых исключительных случаях получение такого доказательства чрезвычайно трудоёмко. Также иногда применяются на практике *полу-разрешающие алгоритмы*. Например, каким-то перебором ищется данное, удовлетворяющее предикату `P`, никакого доказательства завершаемости не известно, а пользователю достаточно тех случаев, когда объект найден в итоге везения. В таких (редких) случаях имеет смысл применять в системе Agda конструкцию `postulate` («поверь») или `NO-TERMINATION-CHECK` — пропустить проверку завершаемости для данной функции.

[2] В недавних версиях языка Agda имеется ещё одно средство для обеспечения доказательства завершаемости — «типы с размером» (sized types). Данная статья предназначена для вводного чтения, поэтому мы пропускаем эту и многие другие подробности относительно программирования на языке Agda.

(4) Наконец, существуют математические сущности, конструктивное представление которых является философской проблемой. Например: определение действительных чисел и алгоритмический поиск корня непрерывной функции на отрезке (в общем случае).

Далее, в обосновании некоторых алгоритмов, действующих над конечными данными применяются теоремы, для которых не известно конструктивное доказательство. Например, иногда применяется теорема о том, что в кольце с единицей существует максимальный идеал.

Как пример самой простой из сложных проблем для конструктивного доказательства рассмотрим лемму Хигмана о словах [**15**]:

> «для всякой бесконечной последовательности слов w(k) в конечном алфавите существуют номера i, j такие, что w(i) есть подслово в w(j) (то есть получается удалением из w(j) некоторых позиций (может быть, никаких))».

Давно известно (C. Nash-Williams) её короткое и неконструктивное доказательство. И ожидалось, что любое формальное конструктивное доказательство будет много длиннее. Позже конструктивные машинно-проверяемые доказательства этой леммы были получены в системах Coq, Isabelle и, наконец, Agda: [**16**, **17**]. И теперь видно, что доказательство в системе Agda (для случая алфавита из двух букв) занимает примерно 80 строчек программы, если печатать до 70 символов в строке (и не ставить пустых строк). Это всего лишь в 2-3 раза длиннее неконструктивного доказательства.

2.2. Предполагаемый выход в проблемных случаях

В практике программирования символьных вычислений в проекте DoCon проблемные случаи вида (3) и (4) пока не встречались. Даже если ограничиться только не-проблемными вычислительными алгоритмами, то будет программно воплощена (в системе формально доказуемого программирования) очень большая часть вычислительной математики.

В проблемных же случаях имеет смысл вводить аргумент – параметр – ограничение числа шагов алгоритма либо постулировать соответствующее свойство алгоритма.

Может иметь смысл передавать эти постулаты как параметры модуля. Тогда проще будет видеть, в доказательствах каких именно теорем они используются.

Заметим, что можно, вообще, постулировать закон исключённого третьего и дальше пользоваться классической логикой. Но это было бы крайне грубым обхождением со свойствами алгоритмов.

Гораздо правильнее по возможности избегать конструкции postulate и постулирования завершаемости.

В текущем выпуске библиотеки DoCon-A конструкция postulate не употребляется, завершаемость также всегда доказывается, и чем дольше не появятся такие постулирования в проекте, тем лучше.

2.3. Доказательство «от противного»

Оно часто оказывается возможным в конструктивной математике (и в языке Agda). Например — в том случае, когда соответствующее отношение P является алгоритмически разрешимым, то есть приложен алгоритм разрешения для P (выдающий ‘да’ или ‘нет’).

- Функции библиотеки DoCon 2014 года имеют дело лишь с разрешимыми отношениями,
- В библиотеке DoCon-A немало конструктивных доказательств «от противного».

В качестве примера неразрешимого предиката можно привести равенство слов в некоторых группах, заданных конечным набором образующих и соотношений.

2.4. Ещё о проблеме доказательств завершаемости

Как сказано выше, занятие программирования символьных вычислений в алгебре потребовало изобретения подходящего способа записи алгебраической области: зависимые типы. Зависимые типы в свою очередь подвигают к использованию доказуемого программирования, ибо было бы неестественно не воспользоваться такой возможностью.

Возникает вопрос: какому теоретическому представлению о конструктивизме (интуиционизме) соответствует подход к доказуемому программированию, принятый в рассматриваемой библиотеке DoCon-A?

Направления конструктивной логики и конструктивной математики разрабатываются, начиная с 1930-х годов, в исследованиях Л. Брауэра, А. А. Маркова, М. Лёфа и других учёных. Имеется различие в представлениях Л. Брауэра и А. А. Маркова о конструктивных объектах. О нём пишет А. А. Марков [13]:

«В этом пункте конструктивисты разных школ, по-видимому, расходятся друг с другом. Я изложу сейчас свою точку зрения на этот вопрос.

Я считаю возможным применять здесь рассуждение „от противного\, т. е. утверждать, что алгорифм A применим к слову P, если предположение о неограниченной продолжаемости процесса применения A к P опровергнуто приведением к нелепости. Мне известно, что интуиционисты отвергают этот способ рассуждения, так как не считают его „интуитивно ясным\.

...

Нетрудно видеть, что рассмотренный способ доказательства применимости алгорифма дает возможность обосновать следующий способ рассуждения.

Пусть для свойства P имеется алгорифм, выясняющий для всякого натурального числа n, обладает ли n свойством P. Если опровергнуто предположение о том, что ни одно число не обладает свойством P, то имеется натуральное число со свойством P. Найти это натуральное число можно тогда путем перебора натуральных чисел начиная с нуля, причем для каждого рассматриваемого натурального числа n мы выясняем, пользуясь алгорифмом, наличие которого предполагается, обладает ли n свойством P. В силу этого мы называем этот способ рассуждения методом конструктивного подбора.»

Как видно из этой статьи, все согласны с тем, что конструктивный объект должен быть итогом работы алгоритма. Но имеется расхождение относительно способов доказательства завершаемости алгоритма. А. А. Марков пишет, что предлагаемое им определение доказательства зваершаемости ведёт к расширению класса конструктивных объектов и лучше соотносится с практическим применением алгоритмов. Это правило А. А. Маркова означает допустимость классических (неконструктивных) доказательств завершаемости алгоритмов.

Библиотека DoCon-A использует язык Agda, что неизбежно подставляет в качестве основания интуиционистскую теорию типов М. Лёфа [14]. Для текущей версии библиотеки DoCon-A достаточно простейшего способа доказательства завершаемости: введением счётчика шагов и применением известной верхней оценки числа шагов для каждого данного метода.

Но в будущем может понадобиться исследовать для практических целей возможность выражения на языке Agda вышеописанного подхода «завершаемость как отрицание бесконечности работы алго-

ритма». Ибо не для всех завершающихся методов имеется прямое доказательство завершаемости.

Как недавно выяснилось из переписки в электронном списке пользователей системы Agda, правило Маркова может быть выражено на яыке Agda через некий особый способ применения постулирования завершаемости, и при этом сохраняется возможность программировать доказательства свойств соответствующего алгоритма.

3. Итог: о преимуществах доказуемого программирования с зависимыми типами

Теоретической основой вычислений с зависимыми типами является конструктивная теория типов Мартина Лёфа [14].

Как только применены зависимые типы, алгоритмы (программы) естественным образом соединяются с *доказательствами*. И появляется возможность *доказуемого программмирования*, когда заданные свойства алгоритма автоматически проверяются компилятором (точнее — проверяльщиком типов). Более определённо, зависимые типы дают возможность:

(1) выражать свойство P алгоритма в виде типа T (зависящего от значений), причём конструкторы для T задаются программистом,
(2) выражать доказательство свойства P в виде функции, строящей любое значение в T,
(3) соединять в исходной программе алгоритм и доказательства его важнейших свойств (выбранных программистом), причём так, что наличие доказательств не замедляет вычисления,
(4) полагаться на автоматическую проверку доказательств,
(5) строить и надёжно автоматически проверять многие доказательства теорем в математике (ибо утверждения выражаются в виде зависимых типов).

Добавим: последние два пункта важны ещё и потому, что

- почти в каждом учебнике есть опечатки и ошибки,
- ошибка в программе, управляющей прибором, может вызвать тяжёлые последствия.

Перечислим некоторые важные черты формальных доказательств в языке Agda.

(1) Проверяльщик типов не пропустит ошибочное или неполное доказательство.
(2) Замечательно, что доказательства являются *данными* языка Agda (и этим иногда бывает выгодно пользоваться).
(3) Проверяльщик типов по умолчанию выполняет поиск доказательства путём сведения (normalization) выражений типов согласно функциям в области видимости. Например, тип `2 + 3 ≡ 3 + 2` сводится по библиотечным функциям к типу `5 ≡ 5`, а свидетельством в последнем типе является стандартный конструктор `refl`, и это есть автоматическое доказательство.

    Доказательство сведением часто оказывается достаточным, но еще чаще бывает недостаточным.
(4) Конструкцией `postulate` программист может заменить доказательство, которое ему трудно составить. Это значит «докажу когда-нибудь потом, а пока поверь на слово». Даже если все доказательства пропустить, получится программа, в которой области вычисления записаны более адекватно, чем это возможно в языках без зависимых типов.
(5) Имеются некоторые функции из стандартной библиотеки, которые облегчают построения доказательств.
(6) Желательно усилить библиотеку таким доказывателем (написанным на языке Agda), который более существенно облегчит построение доказательств.
(7) Доказательство в Agda-программе является полностью формальным, его длинный список тривиальных шагов вывода предназначен для автоматического проверяльщика. А чтение и восприятие его человеком возможно чаще всего на уровне формулировок крупных частей — лемм. Леммы задаются в виде функций, а формулировка леммы представлена объявлением типа функции (signature) и комментарием.

3.1. Пример: программа для упорядочения списка

Эта программа (функция) имеет добавочный аргумент `(<)` для сравнения пары элементов. Обыкновенный подход в системе с зависимыми типами состоит в том, что

(1) записывается в виде конструкторов типа определение того, что отношение `_<_` является полным порядком,
(2) записывается в виде конструкторов типа определение упорядоченности любого списка `xs`,

(3) записывается алгоритм упорядочения `sort`,
(4) записывается в виде типа то свойство, что `sort _<_ xs` возвращает упорядоченный список, обладающий тем же много-множеством (multiset), что и `xs`,
(5) в программу ставится доказательство того, что (1) влечёт (4).

Заметим, что в языках без зависимых типов свойство порядка `_<_` не может быть выражено, компилятором не проверяется, и потому в случае неудачного задания функции `_<_` итог вызова `(sort _<_ xs)` может оказаться неупорядоченным списком.

3.2. Перевод доказательства

В этой статье выражение «составить доказательство» не означает «изобрести доказательство».

Оно означает: *перевести строгое конструктивное «человеческое» доказательство в полное и формальное доказательство на языке Agda.*

«Человеческое» доказательство часто берётся из учебника или научной статьи. Правда, в некоторых случаях понадобится *изобретение* конструктивного доказательства при переводе неконструктивного участка доказательства.

4. Состояние проекта DoCon-A. Общее обсуждение итогов

Цель проекта описана в Разделе 0.3. К июню 2014 года конструктивно-доказательно выражена часть библиотеки DoCon. Эта часть мала (по охвату множества методов) в сравнении с библиотекой DoCon, но существенна — для проверки доказуемо-конструктивного подхода в программировании математики. Именно, сделано следующее.

- Выражена иерархия классических категорий алгебры — от разрешимого множества `DSet` и полугруппы `Semigroup` до евклидова кольца `EuclideanRing` и поля `Field`.
- Расширенный алгоритм НОД запрограммирован для произвольного евклидова кольца.
- Структура (‘случай’, instance) евклидова кольца задана для кольца `Integer` целых чисел.
- Задан конструктор (функтор) области остатков `Q = R/(b)` для произвольного евклидова кольца `R`, с арифметикой в `Q` включающей частичное деление.

- Головная функция `'main'` исполняет арифметический тест для области `Q = R/(b)` при `R = Integer`.
- Все классические определения (свойства операций) выражены в виде (семейств) типов, все необходимые доказательства (без пропусков) включены в программу.
- Скорость вычисления близка к скорости вычисления в системе DoCon (исполняемой под системой Glasgow Haskell).

Мы надеемся сделать первый выпуск библиотеки в 2014-м году.

Объясним некоторые подробности.

4.1. Вычислительная стоимость доказательства

Доказательства не замедляют исполнение программы — при разумном способе её составления. Но они занимают а) объём исходного текста программы, б) память и время на этапе проверки типов.

Некоторые предосторожности бывают нужны, чтобы избежать неоправданных затрат при проверке типов.

В тестовом примере (апреля 2014 года) проверка типов библиотеки DoCon-A длилась 10 минут — на компьютере частоты 2 GHz, и требовала 9 Gb памяти (для системы Agda от февраля 2014).

Многие улучшения возможны в проверяльщике типов, при которых он станет работать быстрее.

4.1.1. *Объём доказательства*

Что замечательно: в (довольно содержательном) примере библиотеки DoCon-A (апреля 2014 года) оказалось, что

> *объём исходного кода для доказательств приблизительно в 5 раз больше, чем объём текста учебника, содержащего все необходимые для этой библиотеки строгие «человеческие» определения и доказательства.*

Это по-видимому значит, что при небольшом развитии библиотеки доказывателя объём исходного кода для формальных доказательств в системе Agda окажется таким же, как объём текста соответствующих глав учебника.

Стандартная библиотека языка Agda содержит реализации доказывателей для некоторого количества специализированных языков.

Например — `EqReasoning` (для записи рассуждений по транзитивности). А также реализации некоторых простых решателей. Например, `RingSolver`.

Есть и статьи по поводу реализации решателей на языке Agda, например, [18].

Упомянем некоторые доказательства, содержащиеся в коде библиотеки DoCon-A апреля 2014 года.

- Определяется понятие группы, доказывается единственность обратного элемента, доказывается равенство $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$; определяется понятие кольца, кольца с разложением на множители, и так далее;
- доказывается равенство Безу для расширенного НОД, определяются арифметические действия в кольце остатков `Q = E/(b)`, доказывается, что это есть коммутативное кольцо;
- доказывается, что для простого `b` область `Q` есть поле;
- определяется область `Integer` целых чисел, доказывается, что её действия удовлетворяют законам евклидова кольца;
- задаётся головная функция для проверочных вычислений в `Q` с подстановкой `E = Integer`.

Все формальные доказательства в программе довольно хорошо читаются, так как применяется такое-же разбиение на леммы, как в учебниках по алгебре.

4.2. Средства перевода доказательства

Программист «переводит» доказательство из учебника в формальное доказательство на языке Agda.

Удивительно: по текущему опыту проекта DoCon-A оказалось, что всего только трёх приёмов достаточно для построения удовлетворительно выглядящего доказательства:

(1) сведение (normalization),
(2) композиция функций,
(3) рекурсия.

Здесь (1) есть доказательство прямым вычислением (упрощением) к одному выражению, (2) соответствует введению леммы, (3) соответствует доказательству индукцией по построению данного.

Объём получаемого доказательства выглядит естественным. Но перевод конструктивного доказательства в полное — формальное обычно требует (в нынешней системе Agda) больших усилий и

времени. Это противоречит нашему ожиданию того, что перевод конструктивного доказательства есть действие по большей части механическое.

Будущее развитие доказывателя как части библиотеки языка Agda должно оправдать это ожидание. Чем мощнее доказыватель, тем легче составлять доказательства на языке Agda.

Пример возможной функции доказывателя: Пусть для векторов u, v_1, v_2, v_3, v_4 четырёхмерного векторного пространства надо поставить в программу доказательство того, что u принадлежит линейной оболочке векторов v_i. Включение в доказыватель алгоритма решения линейной системы даст автоматическое построение соответствующего доказательства.

Добавим, что изобретение доказательства в его содержательной части остается (пока) за человеком, обычно это есть изобретение лемм и разбиение утверждения на леммы. Задача для доказывателя есть заполнение очевидных (для человека) мест подробностями формального доказательства.

Пример: Пусть функция `rev` переворота списка задана рекурсивно через функцию `(++)` соединения списков:

```
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

rev []       = []
rev (x :: xs) = (rev xs) ++ (x :: [])
```

Здесь ‘`::`’ означает приставку элемента к голове списка.

Доказать утверждение: *для любого списка* `xs` `(rev (rev xs) ≈ xs)`.

Это утверждение может быть выражено на языке Agda как объявление типа функции:

```
rev-rev=id : {A : Set} → (xs : List A) → rev (rev xs) ≈ xs,
```

где

`rev-rev=id` — имя функции (смотри Раздел 1.1).

`xs` — переменная аргумента этой функции.

`rev (rev xs) ≈ xs` — семейство типов (T), параметризованное ‘индексом’ `xs`. Если ниже в теле функции правильно задать алгоритм построения любого данного из T, то это будет доказательство утверждения.

`{A : Set}` значит, что `A` есть любой тип. Фигурные скобки означают неявный аргумент, то есть этот аргумент может во многих вызовах быть пропущен (проверяльщик типов сам его восстановит).

Автоматический поиск доказательства этого утверждения (как и многих других простых утверждений) есть тяжёлая проблема для современных доказывателей (этот пример стал решаться после того, как разработчики рассмотрели именно его и подправили доказыватель под задачи похожие на этот пример).

Но если человек изобретёт лемму

```
lemma : {A : Set} → (xs : List A) →
                    rev (xs ++ (x :: [])) ≈ (x :: (rev xs))
```

и поставит её в качестве подсказки, то большинство доказывателей быстро найдут доказательство леммы а также вывод целевого утверждения из леммы.

На практике такой приём сократит затрату времени на составление этого участка программы приблизительно с 60-ти до 5 минут.

5. О других системах с зависимыми типами

Язык Aldor [10] обладает зависимыми типами. Но применяется без построения доказуемых программ.

Система Coq [11] использует язык программирования Gallina, обладающий зависимыми типами, и имеет обширную библиотеку. Например, в этой системе построено машинно-проверяемое доказательство теремы о разрешимости всякой группы нечётного порядка [19]. Эта теорема чрезвычайно важна, имеет сложное и длинное доказательство; формальное машинно-проверяемое доказательство также получено с большим трудом. Но его получение всё-таки свидетельствует о плодотворности подхода доказуемого программирования на зависимых типах.

Кроме чистой функциональности и «ленивого» вычисления, Agda имеет то отличие от Coq, что не использует никакого отдельного языка для доказательств. Дальнейший опыт должен прояснить, насколько эти чистота и красота оправданы.

6. Заключение

Описанный выше опыт программирования на языке Agda символьных вычислений в алгебре показывает действенность подхода конструктивной математики и зависимых типов. Главной очередной проблемой области является создание доказывателя, существенно облегчающего построение формальных доказательств.

В намерения автора входят:

(1) испытание зависимых типов в построении библиотеки DoCon-A вне связи с машинно-проверяемыми доказательствами,
(2) развитие доказуемой версии DoCon-A с конструктивными машинно-проверяемыми доказательствами свойств более сложных алгоритмов, например, завершаемости алгоритма базиса Грёбнера, свойств алгоритма разложения многочлена на множители.

Исследование в части (2) должно дать более ясное представление о практических возможностях конструктивной математики.

Автор благодарит рецензента за ценные замечания.

Список литературы

[1] *Haskell 2010: A Non-strict, Purely Functional Language. Report of 2010.* Home page of the Haskell materials, URL http://www.haskell.org. ↑28, 29

[2] S. D. Mechveliani. *Computer algebra with Haskell: applying functional-categorial- 'lazy' programming* // International Workshop CAAP-2001. — Dubna, Russia, 2001, p. 203–211., URL http://compalg.jinr.ru/Confs/CAAP_2001/Final/proceedings/proceed.pdf. ↑28, 29

[3] U. Norell, J. Chapman. *Dependently Typed Programming in Agda*, URL http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf. ↑28, 30

[4] S. D. Meshveliani. *Dependent Types for an Adequate Programming of Algebra* // Electronic Journal CEUR Workshop Proceedings // Intelligent Computer Mathematics. CICM-WS-WiP 2013, Workshops and Work in Progress at CICM. — Bath, UK, 2013. Vol. **1010**., URL http://ceur-ws.org/Vol-1010/paper-05.pdf. ↑28

[5] S. D. Meshveliani. *Experience in an Adequate Programming of Algebra in a Language with Dependent Types* // Polynomial Computer Algebra 2014. Extended Abstracts. — St. Petersburg: VVM Publishing, St. Petersburg, 2014, p. 54–57. ↑28

[6] А. И. Кострикин. Введение в алгебру. Основы алгебры. М.: Наука. Физматлит, 1994. ↑28

[7] Дж. Коллинз, Р. Лоос. Компьютерная алгебра. Символьные и алгебраические вычисления / ред. Б. Бухбергер. М.: Мир, 1986. ↑28, 37

[8] Дж. Давенпорт, И. Сирэ, Э. Турнье. Компьютерная алгебра. М.: Мир, 1991. ↑28

[9] R. D. Jenks, R. S. Sutor et al. Axiom, the Scientific Computation System. New York–Heidelberg–Berlin: Springer-Verlag, 1992. ↑29

[10] S. Watt et al. *Aldor Compiler User Guide.* IBM Thomas J. Watson Research Center, URL http://www.aldor.org. ↑29, 30, 47

[11] *The Coq Proof Assistant.* Homepage of the Coq system, URL http://coq.inria.fr. ↑30, 47

[12] *Agda. A dependently typed functional programming language and its system.* Homepage of Agda, URL http://wiki.portal.chalmers.se/agda/pmwiki.php. ↑30

[13] А. А. Марков. *О конструктивной математике* // Проблемы конструктивного направления в математике. 2. Конструктивный математический анализ. Труды МИАН СССР. — М.–Л., 1962. Т. **67**, с. 8–14., URL http://mi.mathnet.ru/tm1756. ↑31, 37, 40

[14] P. Martin-Löef. Intuitionistic Type Theory: Bibliopolis, 1984. ↑31, 40, 41

[15] G. Higman. *Ordering by divisibility in abstract algebras* // Proceedings of the London Mathematical Society, 1952. Vol. **s3-2**, no. 1, p. 326–336. ↑38

[16] S. Berghofer. *A constructive proof of Higman's lemma in Isabelle* // Types for Proofs and Programs, International Workshop (TYPES 2003). LNCS / S. Berardi, M. Coppo (eds.): Springer-Verlag, 2004. Vol. **3085**, p. 66–82. ↑38

[17] S. Romanenko. *A proof in Agda of Higman's lemma.*, URL https://github.com/sergei-romanenko/agda-miscellanea/tree/master/Higman. ↑38

[18] P. Kokke, W. Swierstra. *Auto in Agda: Programming proof search*, 2014, URL http://www.staff.science.uu.nl/~swier004/Publications/AutoInAgda.pdf., Under preparation for ICFP 2014. ↑45

[19] A. Mahboubi. *The Rooster and the Butterflies* // Intelligent Computer Mathematics. LNAI. — Berlin–Heidelberg: Springer-Verlag, 2013. Vol. **7961**, p. 1–18. ↑47

Рекомендовал к публикации *к.ф.-м.н. С. А. Романенко*

Об авторе:

Сергей Давидович Мешвелиани

Старший научный сотрудник ИПС РАН. Занимается автоматизацией математических вычислений и рассуждений, функциональным программированием. Автор библиотеки вычислительной алгебры DoCon.

e-mail: mechvel@botik.ru

Образец ссылки на эту публикацию:

С. Д. Мешвелиани. *О зависимых типах и интуиционизме в программировании математики* // Программные системы: теория и приложения: электрон. научн. журн. 2014. Т. 5, № 3(21), с. 27–50.

URL: http://psta.psiras.ru/read/psta2014_3_27-50.pdf

Sergei Meshveliani. *On dependent types and intuitionism in programming mathematics.*

Abstract. It is discussed a practical possibility of a provable programming of mathematics basing of the approach of intuitionism, a language with dependent types, proof carrying code. This approach is illustrated with examples. The discourse bases on the experience of implementing in the Agda language of a certain small algebraic library including the arithmetic of a residue domain $R/(b)$ for an arbitrary Euclidean ring R. (*in Russian*)

Key Words and Phrases: intuitionistic mathematics, algebra, dependent types, Coq, Agda, Haskell.