ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №1(32), 2017, с. 3–46

УДК 510.252, 004.432.42

С. Д. Мешвелиани

Программирование вычислительной алгебры на основе конструктивной математики. Области с разложением на простые множители

Аннотация. Статья продолжает публикации автора о подходе к использованию конструктивной математики и применении языка с зависимыми типами для доказуемого программирования вычислительной алгебры.

Получено конструктивное выражение понятия области с *разложением на простые множители* для *моноида* и *кольца* с некоторыми дополнительными свойствами. Описан способ построения машинно-проверяемых доказательств для теорем, связывающих понятия разложения на простые множители в областях различного вида.

Все описываемые построения и доказательства воплощены полностью в виде программы на функциональном языке Agda.

Ключевые слова и фразы: конструктивная математика, алгебра, факторизация, зависимые типы, функциональное программирование, Agda.

Введение

В статье [1] объясняется подход конструктивной математики [2] и применения конструктивной теории типов [3] к программированию математических вычислений. На примерах из вычислительной алгебры описываются главные черты чисто функционального языка Agda [4, 5], воплощающего подход интуиционистской теории типов. Статья [1] является общим введением в предмет — назовём её «Часть I» изложения.

> Как только применены зависимые типы, алгоритмы (программы) естественным образом соединяются с *доказательствами*. И появляется возможность *доказуемого программирования*, когда заданные свойства алгоритма автоматически проверяются компилятором (точнее —

Исследование выполнено в рамках госзадания ФАНО России — НИР «Развитие методов анализа и верификации моделей вычислительных систем и алгебраических объектов на основе средств функционального и логического программирования» (№ г/р АААА-А16-116021760039-0).

© С. Д. Мешвелиани, 2017
© Институт программных систем имени А. К. Айламазяна РАН, 2017
© Программные системы: теория и приложения, 2017

проверяльщиком типов). Более определённо, зависимые типы дают возможность:

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

Изложение опирается на:

- понятия о чисто функциональном программировании, например, на языке Haskell, о “ленивом” способе вычисления,
- сведения о главных чертах языка и системы (proof assistant) Agda [**4**],
- изложение статей [**1**, **6**],
- общие понятия из алгебры [**7**],
- сведения о вычислительной алгебре [**8**, **9**],
- понятие области с разложением на простые множители ([**10**], глава II, параграф 4: «факториальное кольцо»), ([**11**], 2.14. Factorial monoids and rings).

В статье не рассматриваются ни какие-либо особо эффективные способы разложения на простые множители, ни их программное воплощение. Здесь речь идёт о воплощении на формальном языке конструктивной математики преобразования структуры разложения на множители между областями (трёх) различных видов, с построением машинно-проверяемых доказательств.

Вообще, целью первой очереди проекта DoCon-A [**12**] является проверка практической возможности выражения на языке Agda основных понятий и простейших алгоритмов и теорем классической алгебры. Требуется убедиться в том,

(*1*) что выражения в программе во всех подробностях соответствуют смыслу математических определений, взятых из учебников,

(*2*) что эти выражения достаточно кратки,

(*3*) что компилятор способен за небольшое время проверить довольно большую исходную программу (со всеми доказательствами), выражающую содержание некоторого учебника по алгоритмической математике, и составить исполняемую программу,

(*4*) что эта программа исполняется приблизительно так же быстро, как и соответствующая программа, не содержащая доказательств.

Уточнение к части (3).

Например: библиотека DoCon-A 0.04.1 выражает содержание учебника объёмом приблизительно 150 страниц, исходная программа (без комментариев и пустых строк) занимает примерно в 3 раза больше текста — ибо там много подробностей формальных доказательств, которые в учебниках пропускаются как очевидные. И компиляция (проверка типов + порождение кода) этой программы системой Agda-2.5.2 длится 64 минуты на персональном компьютере частоты 3 гГц.

Правда, при этом занимается слишком много памяти, и этот недостаток в реализации языка Agda предстоит исправлять.

0.1. Пример применения машинно-проверяемого доказательства

Первоначально библиотека `DoCon-A` определяла понятие разложения на простые множители только для случая целостного кольца (как это делается в большинстве учебников). Затем автор встретил в переписке упоминание о том, что в некоторых учебниках рассматривается естественное обобщение этого понятия на случай коммутативного моноида с законом сокращения («CC-моноида», в нашем определении). Тогда автор самостоятельно дал все необходимые определения и доказательства для разложения в CC-моноиде и согласования понятий разложения для целостного кольца и для CC-моноида. В части теории это оказалось нетрудно сделать. Но при таком подходе велика вероятность ошибки.

Поэтому был проведён следующий опыт. До того, как сверять свои выводы с каким-либо учебником, автор составил на языке Agda и в среде библиотеки DoCon-A все вышеупомянутые определения и доказательства. Теперь это суть формальные доказательства, с большим множеством подробностей. И то событие, что они были (после различных исправлений) приняты проверяльщиком типов системы Agda, дало уверенность в теоретической и технической правильности разработки.

И затем автор убедился, что в учебнике [**11**] содержатся почти все те же леммы и теоремы.

0.2. О других системах, использующих аппарат зависимых типов

На языке Aldor [13] запрограммирована богатая библиотека алгебры. При этом зависимые типы используются для адекватного представления алгебраических областей.

Широко известна система Coq [14]. Она использует язык программирования Gallina, обладающий зависимыми типами, и имеет обширную библиотеку, включающую также и развитую библиотеку средств поддержки построения доказательств. Например, в этой системе построено машинно-проверяемое доказательство теоремы о разрешимости всякой группы нечётного порядка [15]. Эта теорема важна в алгебре и имеет сложное и длинное доказательство. Имеется также много доказательных программ для различных сложных алгоритмов в математике, исполненных в системе Coq.

В смысле сравнения системы DoCon-A с названными системами отметим следующие особенности её архитектуры.

(*1*) Библиотека DoCon-A основана на
  (1.а) чисто функциональном программировании,
  (1.б) ленивой модели вычисления,
  (1.в) близости применяемого языка Agda к языку Haskell.

(*2*) На языке Haskell автором написаны многие вычислительные методы и некоторый доказыватель, и эти разработки предполагается применить в развитии библиотеки.

Язык Aldor не принадлежит ни семейству (1.а), ни семейству (1.б). Кроме того, он пока не применяется для построения машинно-проверяемых доказательств.

Язык Gallina системы Coq также не принадлежит ни семейству (1.а), ни семейству (1.б).

В данной же статье речь пойдёт о выражении некоторых математических понятий на языке определённого вида.

0.3. Некоторые соглашения

0.3.1. *О словоупотреблении*

В дальнейшем библиотека программ DoCon-A часто называется коротко «библиотека», а выражение «стандартная библиотека» означает стандартную библиотеку языка Agda.

Слово «доказыватель» используется как перевод английского слова prover.

Часто в этой статье ставится слово «воплощение» вместо слова «реализация» (implementation).

Выражение «берётся из модуля M» заменяет выражение «импортируется из модуля M».

Выражение «разложение на простые множители» часто сокращается до выражения «разложение на простые» или даже до выражения «разложение».

Слово «многомножество» употребляется как перевод слова multiset.

Выражение вида «модель (алгебраической) структуры S» означает любую модель общей алгебраической структуры S — в смысле модели её теории. Это приблизительно соответствует понятию «instance of the class S» в языке программирования, в котором есть классы и их *случаи*. Например, если `Group` есть общая структура группы (определение группы вообще), то группа `int+group` целых чисел по сложению есть модель структуры (класса) `Group`.

0.3.2. *О синтаксисе, символах и именах в языке Agda*

(повтор отрывка из [1]).

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

Agda

```
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.

Дадим ещё несколько объяснений о синтаксисе и смысле конструкций языка [4]:

- Выражение `x : T` означает, что значение `x` имеет тип `T` (а множество-носитель алгебраической области выражается в библиотеке в виде типа).
- `S → T` есть тип всех функций из типа `S` в тип `T`.
- Выражение вида `(\x → f x)` обозначает безымянную функцию от `x`.
- Выражение для типа наподобие `∀ (x y) → P x y` заменяется компилятором на выражение `(x y : T) → P x y`, где `T` есть подходящий тип, выведенный из применения оператора `P`.
- Фигурные скобки, окружающие аргументы функции или конструктора типа, обозначают скрытые аргументы (hidden arguments). Например, для функции сигнатуры

  $$\mathrm{f} : (\mathrm{m} : \mathbb{N}) \to \{\mathrm{n}\ \mathrm{k} : \mathbb{N}\} \to (\mathrm{l} : \mathbb{Z}) \to \mathbb{N}$$

  возможен вызов `(f 0 (+ 2))` — в тех (многих) случаях, когда проверяльщик типов сам выводит значения `n` и `k` из окружения.

Примеры записи формальных доказательств в Agda-программе и записи различных общих алгебраических структур имеются в [1, 6]. Ещё два примера даны ниже.

0.4. Структура статьи

В Разделе 1 описывается башня представленных в библиотеке категорий классической алгебры.

В Разделе 2 описывается представление моноида классов взаимной делимости.

В Разделе 3 описывается представление многомножества (multiset) и его связь с разложением на простые множители.

В Разделах 5 и 6 описывается представление понятия разложения на простые множители в произвольном CCId-моноиде (коммутативный моноид с законом сокращения и вырожденным отношением взаимной делимости).

В Разделе 7 описывается алгебраический приём перенесения понятия разложения на простые (и самой структуры разложения) между областями трёх видов: целостное кольцо–CC-моноид–CCId-моноид. Этот подход есть особая разработка системы `DoCon-A`.

В Разделе 8 описывается представление понятия разложения на простые множители в любом CC-моноиде (коммутативный моноид с законом сокращения), однозначности разложения и связи однозначности со свойством расщепления делимости на простой элемент.

В Разделе 9 описывается представление понятия разложения на простые в целостном кольце, понятия кольца с однозначным разложением на простые, способы получения формальных конструктивных доказательств различных утверждений о таком разложении. Разбирается случай евклидова кольца.

В Разделе 10 описывается, как общие построения из предыдущих разделов согласуются с представлением разложения на простые для областей натуральных и целых чисел.

В Разделе 11 рассматривается пример способа решета разложения натурального числа на простые и некоторая замечательная особенность соответствующей Agda-программы, которая необходима для согласования с необходимостью конструктивного доказательства. Это построение касается частичной замены в языке Agda известного принципа Маркова в конструктивной математике.

Последние два раздела содержат выводы, описание итогов исследования.

1. Башня общих алгебраических структур

В настоящее время библиотека DoCon-A содержит определения и алгоритмы для некоторой иерархии классических алгебраических структур, от разрешимого множества и полугруппы до евклидова кольца и поля. Эта иерархия изображена на рис. 1.

Конечно, пользователь библиотеки волен расширять эту иерархию, программируя дополнительные структуры.

Нисходящие рёбра в этом графе означают наследование структур. Например, полугруппа должна иметь структуру магмы (множества с двуместной операцией) и ещё иметь добавочную структуру, выражающую ассоциативность операции. Два ребра, входящие в вершину `Ringoid` (пред-кольцо) выражают то, что пред-кольцо есть коммутативная группа по сложению (`_+_`) и магма по умножению (`_*_`).

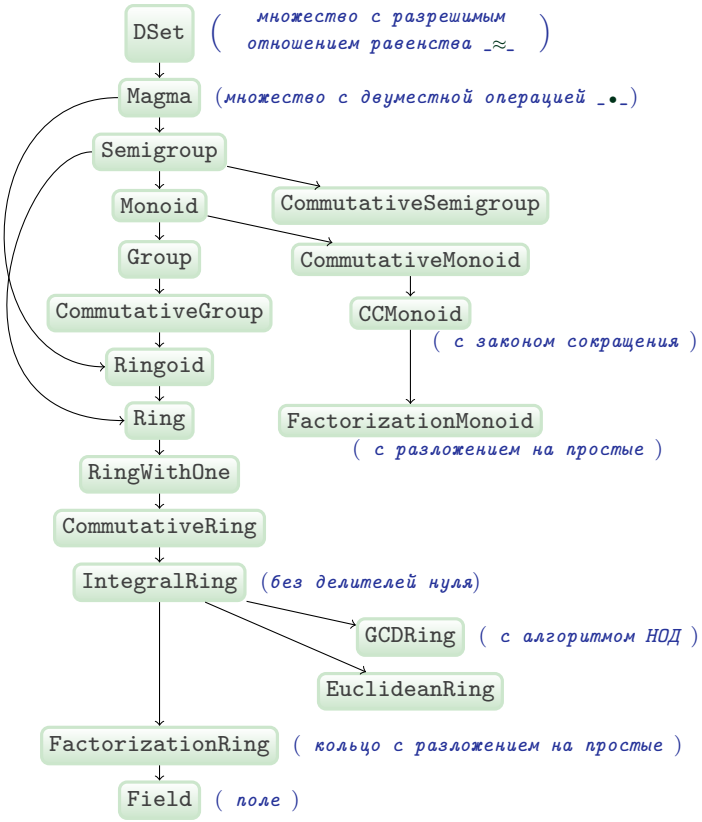

Рис. 1. Башня классических общих алгебраических структур, определённых в библиотеке DoCon-A 0.04.1

В этом графе пропущены некоторые законы, входящие в определения структур и в программу, например, закон дистрибутивности для кольца.

Ниже все алгебраические структуры имеют в основании структуру сетоида (`Setoid`), определённую в стандартной библиотеке как носитель (тип) `Carrier` и отношение равенства _≈_, которое для каждой модели сетоида должно задаваться пользователем ([5], Standard library). Более того, в основе находится разрешимый сетоид (`DecSetoid`), то есть добавлена операция $_\stackrel{?}{=}_$ разрешения для отношения _≈_.

Далее в отрывках программ употребляются обозначения `Monoid`, `UpMonoid`, `Ring`, `UpRing`, и так далее. По математическому смыслу up-<структура> и <структура> не отличаются, а техническое различие объясняется в Разделе 6 статьи [6].

2. Классы взаимной делимости

Понятие простоты элемента алгебраической области и разложения на простые множители связаны с отношением делимости.

Для любой полугруппы `H` библиотека определяет понятия *частного* и отношение делимости `_|_` в `H` в виде объявления типов:

Agda

```
RightQuotient : C → C → Set _
RightQuotient a b =  ∃ \q → (b • q) ≈ a

_|_ : Rel C _
x | y =  RightQuotient S _•_ y x
```

Здесь `C` обозначает тип множества-носителя полугруппы, `_≈_` есть отношение равенства на носителе полугруппы, `_•_` есть операция полугруппы (согласованная с равенством `_≈_`). Эти построения (`C`, `_≈_`, `_•_`) составляют определение библиотечной структуры `Magma`, которую наследует библиотечная структура `Semigroup`.

Конструктор `∃` типа обозначает в стандартной библиотеке конструктивное существование. Например, для натуральных чисел рассмотрим пару

```
(2 , 3*2≡6),
```

в которой вторая составляющая обозначает свидетельство (конструктивное доказательство) равенства `3 * 2 ≡ 6`. Эта пара выражает то, что 2 есть частное для деления натурального числа 6 на 3, и эта пара имеет тип

```
∃ \q → 3 * q ≡ 6.
```

Для введения осмысленных конструктивных понятий простоты элемента и разложения на простые требуется алгоритм распознавания делимости в данной области. Это свойство полугруппы выражается наличием функции `_|?_` соответствующего типа:

Agda

```
_|?_ : Decidable₂ _|_
```

Для рассуждений об однозначности разложения необходимо сначала рассмотреть отношение взаимной делимости. Например, `6 = 2 * 3 = -2 * -3` суть два разложения целого числа. И они должны считаться одинаковыми — в том смысле, что каждое простое из первого разложения делит некоторое простое из второго разложения, и обратно.

Для полугруппы с носителем `C` отношение взаимной делимости определяется объявлением

Agda

```
_~_ : Rel C _
x ~ y = x | y × y | x
```

— то есть `x` и `y` делят друг друга.

Здесь `_×_` есть конструктор произведения типов — он выражает конструктивную логическую связку конъюнкции. Нетрудно доказать, что отношение `_~_` обладает свойствами эквивалентности. Библиотека записывает это доказательство (на языке Agda).

Носитель коммутативного моноида `M`, профакторизованный по отношению `_~_`, в библиотеке имеет имя `Associated`. Это есть множество классов взаимной делимости. Каждый такой класс задаётся выражением `(asd a)`, где `a` есть любой элемент `M`.

Для случая, когда `M` обладает законом сокращения, функции библиотеки доказывают, что отношение взаимной делимости совпадает с отношением *ассоциирования* (association classes): то есть свойство `x ~ y` равносильно в этом случае тому, что `x` и `y` отличаются только некоторым обратимым множителем.

Обычно классы ассоциированных элементов рассматриваются в учебниках для случая целостного кольца (`IntegralRing` из башни рис. 1). Но как описано выше, это определение легко обобщается на случай коммутативного моноида с законом сокращения.

Для коммутативного моноида с разрешённой делимостью библиотека строит структуру *коммутативного моноида* на множестве `Associated`, сопровождая её соответствующими доказательствами.

Далее, функции библиотеки доказывают идемпотентность конструктора `Associated`. Это значит, что в моноиде `Associated` отношение `_~_` вырождено, то есть в области `Associated` отношение `_~_` включено

в отношение _≈_. Библиотека использует это свойство для упрощения сравнения разложений элементов области `Associated`. Например, для двух вышеприведённых разложений целого числа `6` простые элементы `(asd 2)` и `(asd -2)` равны в смысле сетоида области `Associated` для целых чисел.

3. Многомножество

Формальное определение разложения на простые множители требует определения понятий 1) ассоциированного списка пар «ключ–значение» (association list) и 2) многомножества (multiset).

Это необходимо по той причине, что, например, два разложения

Agda

```
18 =  2^1 * 3^2 =  3^2 * 2^1
```

представлены разными списками пар

`((2 , 1) :: (3 , 2) :: [])` и `((3 , 2) :: (2 , 1) :: [])`,

которые, однако, должны задавать одно и то же разложение.

Для сетоида `Key` с разрешимым равенством для «ключей» и любого типа `V` для «значений» ассоциированный список `AssocList` библиотека объявляет как список пар типа `Key × V`, в котором ключи не повторяются в смысле равенства _≈_. Условие отсутствия повторов выражено в виде типа `¬ AnyRepeats`, задание которого мы здесь опускаем.

Многомножество определяется как ассоциированный список со значениями в типе ℕ натуральных чисел, с условием положительности всех значений (вторых частей пар). Значения из `V = ℕ` в многомножестве называем *кратностями* ключей.

Равенство `_=ms_` многомножеств определяется как равенство множеств пар, задаваемых списками пар этих многомножеств. Например, выполнено равенство

Agda

```
MS ((2 , 1) :: (3 , 2) :: [])  =ms  MS ((3 , 2 ) :: (2 , 1) :: [])
```

В библиотеке определяется сложение `_+ms_` многомножеств как объединение ассоциированных списков, со сложением кратностей для возникающих повторов ключей.

Функции библиотеки доказывают, что многомножества относительно равенства `_=ms_` и операции `_+ms_` составляют *коммутативный моноид*.

Эта теорема необходима для доказательства многих свойств структуры разложения на простые. И она очевидна с математической точки зрения. Но составление её формального доказательства трудоёмко. Оно потребовало введения большого количества вспомогательных функций, лемм и индуктивных доказательств, касающихся списков ключ-значение. Пример такой леммы: операция `_+ms_` ассоциативна (относительно равенства `_=ms_`).

Пусть область ключей `Key` есть некоторый коммутативный моноид `Mon`. Например, это так в случае, когда многомножество представляет разложение на простые в целостном кольце. Для такого случая определяется функция *значения* многомножества в моноиде:

Agda

```
msValue M = Π [p ^ e | (p , e) ∈ msPairs M]
```

— каждое многомножество `M` отображается в элемент моноида, являющийся произведением степеней пар для списка пар из `M`. Здесь `_^_` есть возведение в степень в моноиде `Mon`. В библиотеке доказывается, что операция `msValue` есть гомоморфизм из моноида многомножеств в моноид `Mon`.

Библиотека содержит под-библиотеку AList, воплощающую в виде доказательных программ многочисленные операции над списками вообще, ассоциированными списками, многомножествами. AList есть библиотека общего назначения, она не касается ничего существенно алгебраического и по смыслу могла бы быть добавлена к стандартной библиотеке.

Заметим, что накопление подобной под-библиотеки общего назначения, вообще, сильно облегчает построение дальнейших доказательств в алгебре.

Здесь мы опускаем описание каких бы то ни было подробностей построения библиотеки AList.

4. Отступление: примеры доказательств на языке Agda

4.1. Пример: монотонность сложения натуральных чисел

Натуральные числа типа ℕ записываются в унарной системе, через конструктор данных `suc` — «следующий». Например, число `3` представлено в виде `(suc (suc (suc 0)))`.

В исходной программе эти числа можно задавать литералами `0, 1, 2, 3 ...`, но они преобразуются в унарное представление.

Стандартная библиотека даёт следующее определение сложения натуральных чисел:

Agda

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

Agda сама распознаёт завершаемость этого алгоритма, так как во втором предложении функция _+_ вызывается справа на синтаксически меньшей паре аргументов, чем слева.

Отношение «меньше или равно» для натуральных чисел определено в стандартной библиотеке в виде такого типа данных:

Agda

```
data _≤_ : Rel ℕ Level.zero
   where
   z≤n :  ∀ {n}                → 0 ≤ n
   s≤s :  ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n
```

Напомним, что здесь m≤n есть имя переменной, а m ≤ n есть выражение для типа с конструктором ≤, знак которого отделён пробелами.

Это определение означает, что свидетельства (доказательства) для неравенств есть данные этого типа, и они строятся применением конструкторов z≤n и s≤s.

Первое правило (аксиома) означает, что z≤n является свидетельством для утверждения 0 ≤ n для любого n.

Второе правило означает, что если m≤n есть свидетельство неравенства m ≤ n, то (s≤s m≤n) есть свидетельство для неравенства suc m ≤ suc n.

Например, в программе

Agda

```
p :  2 ≤ 5
p =  s≤s (s≤s z≤n)
```

значение p является свидетельством для неравенства 2 ≤ 5. Причём числовые аргументы в выражении для p пропущены, они являются скрытыми (Раздел 0.3.2), в данном примере проверяльщих типов сам их восстанавливает, исходя их объвления типа для p. И получится выражение

Agda

```
p = s≤s {1} {4} (s≤s {0} {3} (z≤n {3}))
```

Если же проверяльщих типов сообщает о неспособности определить тип, то следует проставить некоторые из скрытых значений.

Следующая программа есть доказательство утверждения о монотонности сложения: для любых k, m, n (если $k{\le}m$, то $n+k{\le}n+m$).

Agda

```
open import Data.Nat using (ℕ; suc; _+_; _≤_; s≤s)

  monot+ : (k m n : ℕ) → k ≤ m → n + k ≤ n + m
  monot+ _ _ 0     k≤m = k≤m
  monot+ k m (suc n) k≤m = s≤s n+k≤n+m
                    where
                    n+k≤n+m : n + k ≤ n + m
                    n+k≤n+m = monot+ k m n k≤m
```

Доказательство в данном примере есть функция, которая из значений аргументов строит какое-либо значение в типе `n + k ≤ n + m`.

В первой строке кода разбирается случай `n = 0`. И целью является выдать значение типа `0 + k ≤ 0 + m`.

Проверяльщик типов нормализует (вычисляет в символьном виде, на этапе проверки типов, запуская внутренний интерпретатор) выражения типов согласно определениям функций. Типы, нормализующиеся в одно выражение, считаются по смыслу программы одинаковыми. В данном примере выражение типа `0 + k ≤ 0 + m` нормализуется в выражение `k ≤ m`. Значение `k≤m` имеет тип `k ≤ m`. Следовательно оно также имеет тип `0 + k ≤ 0 + m`. Это значит, что для случая `n = 0` доказательство получено.

Следующая строка разбирает случай, когда третий аргумент есть `(suc n)`. Здесь целью является построить любое значение в типе

(1) `(suc n) + k ≤ (suc n) + m`.

Значение `n+k≤n+m` является свидетельством неравенства `n + k ≤ n + m`, Оно строится рекурсивным вызовом функции `monot+`, в котором аргумент `n` синтаксически меньше значения `(suc n)`.

Далее, значение `(s≤s n+k≤n+m)` имеет тип `suc (n + k) ≤ suc (n + m)`. По правилу нормализации этот тип и тип `(suc n) + k ≤ (suc n) + m` совпадают, поэтому значение `(s≤s n+k≤n+m)` является свидетельством для целевого неравенства (1).

4.1.1. *О смысле функции-доказательства*

Сама исходная программа `monot+` является общим доказательством свойства монотонности операции `_+_` (для любых `k, m, n`). Она представляет собой доказательство по индукции по построению числа `n`. Компилятор воспринимает её как общее доказательство, где `k, m, n` являются символами, переменными, и в таком виде он проверяет это доказательство.

Задание и смысл доказательств в программе полностью соответствует написанию и смыслу конструктивных доказательств в учебниках и научных статьях, с тем отличием, что доказательства в программе полностью формальны и содержат все подробности. Проверяльщик типов сам указывает на недоказанное место. Такое указание всегда имеет вид «не вижу, почему это значение (`p`) имеет такой тип (`T`)» (когда `p` строится как доказательство для утверждения `T`).

Различные разрабатываемые библиотеки доказывателей призваны помочь программисту заполнять подробности доказательств.

4.2. Пример: задание полугруппы натуральных чисел

Этот пример взят из статьи [1].

Модель полугруппы натуральных чисел по сложению задаётся такой программой:

Agda

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

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

В качестве полугрупповой операции в модель подставляется сложение `_+_`, которое задано в предыдущем примере.

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

Во втором предложении правая часть является доказательством 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, отрывок кода на предыдущей странице

Agda

```
12                    =n[ PE.cong suc (assoc+ x y z) ]
```

означает, что применяется закон ассоциативности к подвыражению `(x + y) + z`, а потом применяется закон конгруэнтности операции `suc` относительно равенства `_=n_`.

5. Коммутативный моноид с законом сокращения

В любой полугруппе закон сокращения (слева) определяется так:

Для любых `x, y, z` из `H` если `x • a = x • b`, то `a = b`.

На языке Agda он записывается в виде объявления типа:

Agda

```
1 LCancellation : Set _
2 LCancellation = ∀ x y z → (z • x) ≈ (z • y) → x ≈ y
```

Определение 1. *Коммутативный моноид с законом сокращения и разрешимым отношением деления будем кратко называть CC-моноидом (cancellative commutative monoid).*

5.1. Вырожденное отношение взаимной делимости

Очевидно, имеет место включение отношений $_\approx_ \subseteq _\tilde{}_$.

Определение 2. *Двуместное отношение* $_\tilde{}_$ *на сетоиде назовём вырожденным, если оно включено в отношение* $_\approx_$ *равенства этого сетоида.*

Определение 3. *CC-моноид с вырожденной взаимной делимостью будем кратко называть CCId-моноидом.*

CCId-моноиды занимают особое место.

Для CCId-моноида свойства разложения на множители выражаются и доказываются проще. В таком моноиде единица (ε) есть единственный обратимый элемент.

Приведём примеры:

(*1*) Мультипликативный моноид $\mathbb{N}\backslash 0$ является CCId-моноидом. Ибо если ненулевые натуральные `m` и `n` делят друг друга, то $\mathtt{m} \equiv \mathtt{n}$.

(*2*) Как сказано выше, для любого CC-моноида моноид `Associated` классов взаимной делимости имеет вырожденное отношение взаимной делимости.

(*3*) На мультипликативных моноидах $\mathbb{Z}$ и $\mathbb{Z}\backslash 0$ отношение `_~_` взаимной делимости невырождено, например, `1 ~ -1` и $\mathtt{1} \not\equiv \mathtt{-1}$.

(*4*) На мультипликативном моноиде `Associated` для $\mathbb{Z}\backslash 0$ отношение `_~_` вырождено, как это следует из (2).

(*5*) Для кольца `Q[x]` многочленов от одной переменной над полем рациональных чисел имеется много различных классов взаимной делимости `((asd x), (asd x^2), ...)`, причём всякий ненулевой класс является счётным множеством.

6. Понятие разложения на простые в CC-моноиде с вырожденной взаимной делимостью

Понятие простоты элемента и структура (понятие) факторизации в CCId-моноиде в библиотеке DoCon-A определяется в рамках некоторой среды: параметрического модуля. Заголовок этого модуля

Agda

```
module OfCCMon-ofIdAsd-0
  (upCMon : UpCommutativeMonoid _)
  (open UpCommutativeMonoid upCMon
              using (setoid; _•_; _˜asd_; IdenticAsd))
  (lCancel : LCancellation setoid _•_)
  (˜asd⊆≈ : IdenticAsd)
```

означает, что все построения в этом модуле определены для призвольного CC-моноида `upCMon` с вырожденной взаимной делимостью `_˜asd_`; что функция `lCancel` есть свидетельство закона сокращения, а функция `˜asd⊆≈` есть свидетельство вырожденности отношения `_˜asd_`.

В этой среде выражены понятия простого элемента и разложения на простые множители. Понятие простого элемента задаётся в виде типа

Agda

```
IsPrime : C → Set _
IsPrime p =  p ≉ ε  ×  (∀ {x y} → (x • y ≈ p) → x ≈ ε ⊎ y ≈ ε)
```

Здесь ε есть единица моноида, `p` $\not\approx \varepsilon$ есть отрицание равенства `p` $\approx \varepsilon$. Конструктор `_⊎_` разделённой суммы типов выражает конструктивную логическую связку дизъюнкции.

Таким образом выражение типа в правой части означает: «`p` не есть единица, и если `p` есть произведение, то среди сомножителей есть единица».

Определение разложения элемента `a` на простые объявляется в виде *записи* (record) с *зависимыми* полями:

Agda

```
record Factorization (a : C) :  Set _
  where
  constructor factorization'

  field multiset  : Multiset
        allPrime  : All IsPrime (msKeys multiset)
        productEq : a ≈ msValue multiset
  ...
```

Строки, выделенные ключевым словом `field`, составляют собственно определение данной алгебраической структуры, а остальные части (обозначенные здесь троеточием) нужны для задания вспомогательных

значений, с учётом того, что запись используется не только как определение, но и как некоторый набор функций и доказательств лемм, касающихся этого определения. Чтобы воспользоваться такой функцией или леммой, надо применить запись как программный модуль, вызвав значение из этой записи [4].

Поле `multiset` есть собственно разложение — многомножество простых элементов. Это есть конечное множество простых элементов, различных в смысле равенства `_≈_`, в котором каждому элементу приписана его кратность в элементе `a`.

Поле `allPrime` есть аксиома о том, что все ключи многомножества `multiset` суть простые элементы.

Поле `productEq` есть аксиома равенства

Agda

```
a ≈ msValue multiset
```

Здесь `a` равно значению многомножества из разложения в моноиде (то есть произведению степеней пар разложения).

Пример: в моноиде `Associated` для ненулевых целых чисел разложение класса `[-18]` на простые выражается как данное языка Agda в виде

Agda

```
    factorization' M prime-[2][3] [-18]=[2]^1*[3]^2,
```

где:

- многомножество `M` имеет список пар `([2] , 1) :: ([3] , 2) :: []`,
- `prime-[2][3]` есть свидетельства простоты классов `[2]` и `[3]`,
- `[-18]=[2]^1*[3]^2` есть свидетельство равенства
  `[-18] =asd ([2] ^ 1) * ([3] ^ 2)`
  в моноиде `Associated`.

6.1. CCId-моноид с разложением на простые множители

В той-же среде CCId-моноида определяется CCId-моноид с разложением на простые:

Agda

```
record FactorizationMonoid-ofIdAsd : Set _
  where
  field _|?_      : Decidable₂ _|_
        factor    : (a : C) → Factorization a
        prime?    : Decidable IsPrime
        hasSquare? : Decidable HasSquare
```

Иначе говоря, это такой CCId-моноид, который снабжён:

- операцией (алгоритмом) `_|?_` распознавания делимости,
- операцией `factor` разложения на простые,
- операцией `prime?` распознавания простоты элемента,
- операцией `hasSquare?` распознавания наличия кратного множителя.

Заметим, что последние две операции могут быть просто определены при помощи операции `factor`. Но они включены в определение из-за того, что во многих случаях для этих двух функций существуют особые способы вычисления, имеющие меньший порядок сложности, чем операция `factor`.

6.2. Об однозначности разложения на простые множители

В том-же параметрическом модуле `OfCCMon-ofIdAsd-O`, задающем среду CCId-моноида, определяется понятие однозначности разложения на простые:

Agda

```
FactorizationIsUnique : Set _
FactorizationIsUnique =
 ∀ {a b} → a ≈ b →
        (ft  : Factorization a) →
        (ft' : Factorization b) → m ft =ms m ft'
                          where
                          m = Factorization.multiset
```

Правая часть этого объявления представляет свойство:
«любые два разложения на простые для любых равных элементов данной области содержат равные многомножества» (то есть всякая пара из одного многомножества принадлежит и другому многомножеству).

Это есть формализация классического определения «разложения совпадают с точностью до некоторой перестановки».

Не во всех областях с разложением на простые это свойство выполнено.

Есть свойство, которое выражается проще и которое равносильно однозначности разложения: «если простое p делит произведение a • b, то p делит a или p делит b»

Agda

```
Prime|split : Set _
Prime|split = ∀ {p a b} → IsPrime p → p ∣ (a • b) → p ∣ a ⊎ p ∣ b
```

Назовём это свойство «расщепление делимости».

Доказать однозначность разложения означает в нашем случае запрограммировать функцию

Agda

```
factorizationIsUnique : FactorizationIsUnique
```

В библиотеке это утверждение выводится из условия расщепления делимости. Это доказательство дано в модуле UFtMonoidOfIdAsd-0, который задаёт среду CCId-моноида и свойства Prime|split.

6.3. Важнейшие леммы о разложении на простые в CCId-моноиде

Вот ещё два примера утверждений, доказанных в библиотеке для разложении в CCId-моноиде.

Agda

```
primary|split :
 ∀ {p n a b} → IsPrime p → Coprime a b → (p ^ n) ∣ (a • b) →
                                  (p ^ n) ∣ a ⊎ (p ^ n) ∣ b
```

— выведено из условия Prime|split. Здесь (Coprime a b) есть условие взаимной простоты a и b.

Agda

```
•factorization : {a b : C} → Factorization a → Factorization b →
                                  Factorization (a • b)
```

Эта функция строит разложение для произведения a • b из разложений для a и b. При этом, конечно, строятся все свидетельства необходимые в структуре разложения. В частности доказывается, что сумма многомножеств из разложений для a и b является многомножеством для разложения произведения a • b.

7. Перенесение «кольцо–моноид–CCId-моноид»

В библиотеке DoCon-A разработан и применяется приём перенесения понятий и построений, связанных с разложением на простые множители между областями трёх видов:

(*1*) целостным кольцом `R`,
(*2*) мультипликативным моноидом `M\0` ненулевых элементов в `R` (который есть CC-моноид),
(*3*) CCId-моноидом `Assoc-M\0` классов взаимной делимости для `M\0`.

Мы считаем их уровнями сложности понятий и построений для разложения на простые, где “нижний” уровень (3) наиболее прост. Конструкции и доказательства с уровня (3) переносятся сначала на уровень (2), потом на уровень (1).

Между уровнями (1) и (2) перенос устроен посредством отображений

Agda

```
toNZ : R → R\0,    fromNZ : R\0 → R.
```

`fromNZ` есть вложение из области ненулевых элементов; `toNZ` действует как вложение на ненулевых элементах, но отображает 0 в 1.

Между уровнями (2) и (3) перенос устроен посредством отображений

Agda

```
asd : M\0 → Assoc-M\0,   fromAsd : Assoc-M\0 → M\0.
```

`asd` есть проекция на класс, `fromAsd` есть взятие представителя класса. Используется гомоморфность отображения `asd` по умножению.

Ввиду похожести понятий разложения для областей трёх вышеназванных видов, определение простоты элемента во всех трёх случаях обозначено одним именем `IsPrime`, структура разложения на простые обозначена одним именем `Factorization`, и так далее, хотя эти имена для областей (1), (2) и (3) определены несколько по-разному в соответствующих модулях.

8. Понятие разложения на простые в коммутативном моноиде с законом сокращения

Пусть `M` есть CC-моноид (коммутативный моноид с законом сокращения и с разрешимым отношением деления).

Этот случай является более общим по сравнению со случаем CCId-моноида.

Для CC-моноида имеют смысл понятия простого элемента и разложения на простые множители. Понятие простого элемента в данной среде выражено в виде типа

Agda

```
IsPrime : C → Set _
IsPrime p = p ∤ ε × (∀ {x y} → (x • y ≈ p) → x ∣ ε ⊎ y ∣ ε)
```

Здесь оператор `_∤_` есть отрицание для отношения делимости. Таким образом выражение типа в правой части означает: «`p` необратим, и если `p` есть произведение, то среди сомножителей есть обратимый элемент».

От случая CCId-моноида это определение отличается тем, что свойство равенства единице ослаблено до свойства обратимости.

Это есть классическое определение простоты элемента, но в большинстве учебников ([**10**], Глава II, параграф 4) оно даётся для *целостного кольца* (`IntegralRing`). А такое определение не включает случая моноида. Например, есть понятие простого натурального числа, а область натуральных чисел кольца не составляет.

Поэтому естественно рассматривать понятия простоты элемента и разложения на простые в более общем случае: для CC-моноида. При этом оказывается, что вывод различных свойств разложения на простые для CC-моноида легко (с математической точки зрения) переносится на вывод этих свойств для целостного кольца `R`. Связь этих двух построений состоит в том, что мультипликативный моноид `*monoid-R\0` ненулевых элементов является CC-моноидом.

Понятие разложения на простые в CC-моноиде изложено в ([**11**], 2.14). Впрочем, этот предмет прост с математической точки зрения, и читатель мог бы сам дать доказательства всех утверждений о моноидах, приводимых в этой статье — если только речь не идёт о полных формальных доказательствах.

Структура разложения на простые в CC-моноиде в библиотеке определяется в рамках среды — параметрического модуля. Заголовок этого модуля

Agda

```
module OfCancellativeCommMonoid-1 (upCMon : UpCommutativeMonoid)
    (open UpCommutativeMonoid upCMon using (setoid; _•_; _|_))
    (cancel : LCancellation setoid _•_)
    (_|?_  : Decidable₂ _|_)
```

означает, что все построения в этом программном модуле определены для произвольного CC-моноида `upCMon`, что функция `cancel` выражает закон сокращения, а функция `_|?_` есть алгоритм распознавания делимости.

Определение разложения элемента `a` на простые объявляется в виде *записи*

Agda

```
record Factorization (a : C) : Set _
  where
  constructor factorization'
  field
    multiset  : Multiset                    -- многомножество
    allPrime  : All IsPrime (msKeys multiset) -- простота ключей
    productEq : a ~ msValue multiset        -- равество разложения

  aList : AssocList
  aList = msAL multiset

  ftPairs : List (C × ℕ)
  ftPairs = pairs aList      -- список пар  (p , кратность p в a)

  primes : List C
  primes = map proj₁ ftPairs  -- список ключей - простых множителей

  asdPrimes : List Associated
  asdPrimes = map asd primes

  field  noRepetInAsdPrimes :  ¬ AsdL.AnyRepeats asdPrimes
                      -- условие отсутствия повторов ключей
                      -- с точностью до взаимной делимости
  ...
```

Многоточие замещает здесь несколько значений и лемм, которые выводятся из содержимого полей – аксиом. Например, функцию `ftMultiplicity` извлечения кратности данного ключа в разложении.

Поле `multiset` есть собственно разложение, то есть многомножество простых элементов.

Поле `allPrime` есть аксиома о том, что все ключи многомножества `multiset` суть простые элементы.

Поле `productEq` есть свидетельство для равенства

Agda

```
1                a ~ msValue multiset
```

Здесь `a` равно значению многомножества из разложения в моноиде — с точностью до обратимого множителя.

Например, в разложении `(-2 ^ 1) * (3 ^ 2)` для `a = 18` в моноиде ненулевых целых чисел многомножество записывается как `MS ((-2 , 1) :: (3 , 2) :: [])`, а аксиома `productEq` есть свидетельство того, что значения `18` и `(-2^1 * 3^2)` отличаются на некоторый обратимый множитель.

8.1. Область применения определения

Определение предыдущего пункта не имеет смысла для мультипликативного моноида натуральных чисел. Ибо из-за присутствия нуля не выполнен закон сокращения. Однако оно имеет смысл для мультипликативного моноида `ℕ\0` ненулевых натуральных, ибо там закон сокращения выполнен.

Далее, пусть `R` есть целостное кольцо с разрешимым делением. Тогда мультипликативный моноид `R\0` есть CC-моноид, и для него имеют смысл вышеприведённое определение разложения на простые и все относящиеся к этому построения.

Функции библиотеки доказывают это утверждение о CC-моноиде `R\0`, и это доказательство делает применимыми все функции и теоремы из модуля `OfCancellativeCommMonoid-1` к моноиду `R\0`.

Например, таким образом определение разложения на простые и все относящиеся к этому построения выражаются для ненулевых целых чисел, ибо целые числа образуют целостное кольцо.

8.2. Моноид с разложением на простые множители

Также в среде модуля `OfCancellativeCommMonoid-1` определяется понятие моноида с разложением на простые множители:

Agda

```
open OfAssociated upCMon _|?_ using (Associated; asd; asdCancellation; ...)
  renaming (asdUpCommutativeMonoid to asdUpCMon ...)

asdCancel = asdCancellation cancel
open module Asd = OfCCMon-ofIdAsd-0 asdUpCMon asdCancel ~~⊆=’
  using (FactorizationMonoid-ofIdAsd)

record FactorizationMonoid : Set _ where
    constructor ftMonoid’
    field  ftMonoid-ofIdAsd : FactorizationMonoid-ofIdAsd
    ...
```

Поясним это определение.

Значения `upCMon`, `_|?_`, `cancel` суть данные выше в среде модуля `OfCancellativeCommMonoid-1` параметры: соответственно CC-моноид, функция распознавания делимости в нём, свидетельство для закона сокращения в этом моноиде.

Модуль `OfAssociated` воплощает (со всеми необходимыми доказательствами) коммутативный моноид `asdUpCMon` для области `Associated` классов взаимной делимости для моноида `upCMon`. Он также воплощает функцию `asdCancellation` вывода закона сокращения `asdCancel` для моноида `Associated` из закона сокращения `cancel`.

Параметр `~~⊆=’` есть ранее полученное в библиотеке доказательство того, что отношение взаимной делимости в моноиде `Asssociated` включено в отношение равенства `_≈_` на этом моноиде.

Далее, вышеописанному модулю `OfCCMon-ofIdAsd-0` даются соответствующие параметры, и из этого модуля берётся понятие `FactorizationMonoid-ofIdAsd` моноида с разложением для модели CCId-моноида на носителе `Associated`.

Наконец, структура `FactorizationMonoid` содержит только одну операцию — аксиому: `ftMonoid-ofIdAsd` — структура моноида с разложением для модели CCId-моноида для моноида `Associated`.

Она также содержит дополнительные функции `factor`, `prime?`, `hasSquare?`.

Поясним главную из них:

Agda

```
  factor : ∀ a → Factorization a
  factor a = fromAsdFactorization a (factor-asd (asd a))
```

Эта выражение означает, что разложение элемента a моноида upCMon получается из разложения элемента (asd a) моноида Associated применением функции fromAsdFactorization. Разложение в моноиде Associated осуществляется применением операции factor из структуры ftMonoid-ofIdAsd (эта операция берётся с переименованием в factor-asd).

Перенос разложения происходит с помощью гомоморфизма моноидов

Agda

```
                asd : C → Associated,
```

где C обозначает носитель моноида upCMon.

Функция

Agda

```
  fromAsdFactorization : (a : C) → Asd.Factorization (asd a) →
                                   Factorization a
```

воплощена выше в модуле – среде OfCancellativeCommMonoid-1. Для любого a из upCMon она отображает разложение элемента (asd a) моноида Associated в разложение элемента a в моноиде upCMon.

Пример.
Для разложения числа -18 из моноида $\mathbb{Z}\backslash 0$ ненулевых целых чисел сначала разлагается класс (asd -18) в моноиде Associated $\mathbb{Z}\backslash 0$, и получается разложение с многомножеством

Agda

```
            MS ((asd 2 , 1) :: (asd 3 , 2) :: []).
```

Затем функция fromAsdFactorization отображает это разложение в разложение в $\mathbb{Z}\backslash 0$ с многомножеством

Agda

```
            MS ((2 , 1) :: (3 , 2) :: []).
```

8.3. Понятие однозначности разложения для CC-моноида

В вышеописанном модуле OfCancellativeCommMonoid-1 определяется понятие однозначности разложения на простые для произвольного CC-моноида:

Agda

```
FactorizationIsUnique : Set _
FactorizationIsUnique =
  ∀ {a b} → a ≈ b → (ft :  Factorization a) → (ft' : Factorization b) →
      (map pairToAsd (ftPairs ft)) ~pset (map pairToAsd (ftPairs ft'))
```

Здесь единственность разложения выражена как равенство мультимножеств над областью `Associated`. Из разложения `ft` извлекается список пар, отображение `pairToAsd` отображает ключ пары в область `Associated`. То же делается с парами разложения `ft'`. Далее, отношение `~pset` означает равенство *множеств* пар над областью `Associated`, задаваемых полученными списками: ключи сравниваются как элементы сетоида `Associated`, а кратности — как натуральные числа.

Видно, что это определение отличается от определения однозначности разложения в CCId-моноиде. Это различие соответствует определениям классической алгебры.

8.4. Теорема об однозначности разложения для расщепимого деления

Для CC-моноида определение расщепления `Prime|split` деления на простой элемент выглядит так-же, как для CCId-моноида (Раздел 6.2). Но входящее в него условие `(IsPrime p)`, как написано выше, имеет немного более общее определение.

Параметрический модуль с заголовком

Agda

```
module OfUniqueFactorizationMonoid _
    (upCMon : UpCommutativeMonoid _ _)
    (open UpCommutativeMonoid upCMon using (setoid; _•_; _|_))
    (cancel : LCancellation setoid _•_)
    (_|?_   : Decidable₂ _|_)
    (open OfCancellativeCommMonoid-0 upCMon cancel _|?_
        using (Prime|split))
    (open OfCancellativeCommMonoid-1 upCMon cancel _|?_
        using (FactorizationMonoid))
    (ftMon      : FactorizationMonoid)
    (prime|split : Prime|split)
```

содержит формальный вывод однозначности разложения на простые множители в CC-моноиде `upCMon` из условия `prime|split` расщепения деления на простой элемент для этого моноида.

Условие `prime|split` для моноида `upCMon` переносится на такое же условие `prime|split-asd` для моноида `Associated` для `upCMon`. Однозначность разложения в моноиде `Associated` выведена из условия `prime|split-asd` как для CCId-моноида (Раздел 6.2). Далее, однозначность разложения в моноиде `upCMon` выводится из однозначности разложения в моноиде `Associated`.

9. Случай целостного кольца

9.1. Понятие простоты элемента

Параметрический модуль с заголовком

Agda

```
module OverIntegralRing-2 (upIR : UpIntegralRing)
    (_|?_ : let Mg = UpIntegralRing.*magma upIR
        in Decidable₂ (Magma._|_ Mg)
    )
```

задаёт среду произвольного *целостного кольца* `IR` (`upIR` в программе) — коммутативного кольца без делителей нуля (`IntegralRing`), с функцией `_|?_` распознавания делимости.

Постоянные `0#` и `1#` обозначают соответственно нуль и единицу кольца. `C` ниже обозначает носитель (`Carrier`) кольца `IR`. Простой элемент в `IR` определяется предикатом

Agda

```
  IsPrime : C → Set _
  IsPrime p = p ∤ 1# × (∀ {x y} → (x * y ≈ p) → x ∣ 1# ⊎ y ∣ 1#)
```

— «не обратим, и если разлагается в произведение, то среди сомножителей есть обратимый».

Это определение отличается от определения простоты элемента в мультипликативном CC-моноиде `IR\0` только тем, что возможен случай `p = 0#`. Из условия `IsPrime p` легко выводится неравенство `p ≉ 0#`. Но программа на языке Agda имеет дело с формальными обоснованиями, а с формальной точки зрения предикат `IsPrime` определён на носителе, имеющем дополнительный элемент — `0#`.

Понятия простоты, разложения на простые и соответствующие построения и формальные доказательства переносятся между тремя уровнями:

Agda

```
        IR  →toNZ→   Nonzero  →asd→    Associated
           ←fromNZ←          ←fromAsd←
```

Nonzero есть мультипликативный CC-моноид ненулевых элементов из IR. Associated есть CCId-моноид классов взаимной делимости для моноида Nonzero.

Область Nonzero представлена парами вида (x , x≉0), где x : C, а x≉0 : ¬ x ≈ 0# есть свидетельство неравенства нулю.

Отображение toNZ определено как

Agda

```
toNZ : C → Nonzero
toNZ x =  case x ≟ 0# of \  { (yes x≈0) → (1# , 1≉0)
                            ; (no x≉0)  → (x , x≉0) }
```

То есть для x ≈ 0# выдаётся 1' = (1#, 1≉0), а всякий другой элемент отображается в (x , x≉0).

Отображение fromNZ определено как проекция fromNZ (x , _) = x. Отображения toNZ и fromNZ взаимно обратны на множестве IR\0. Отображения fromNZ и asd суть гомоморфизмы моноидов.

Перенос построений между уровнями Nonzero и Associated описан выше в Разделе 8.2.

В модуле OverIntegralRing-2 доказываются леммы

Agda

```
IsPrime→IsPrime-nz :  ∀ {p} → (prime-p : IsPrime p) →
                 let p≉0 = prime→≉0 prime-p
                 in  NZPack0.IsPrime (p , p≉0)

IsPrime-nz→IsPrime :  ∀ {p} → (p≉0 : p ≉ 0#) →
                      NZPack0.IsPrime (p , p≉0) → IsPrime p
```

Первая из них утверждает, что для простого элемента p : C его образ (p , p≉0) в моноиде Nonzero является простым в смысле CC-моноида. Причём в самом утверждении леммы вызывается функция prime→≉0, которая строит свидетельство для условия p≉0 из свидетельства простоты p.

Вторая лемма для ненулевого p строит свидетельство простоты p в кольце IR из свидетельства простоты элемента (p , p≉0) : Nonzero.

9.2. Понятие разложения на простые множители

В той же среде целостного кольца IR с разрешимой делимостью определяется понятие разложения на простые для элемента a из IR:

Agda

```
record Factorization (a : C) : Set _
  where
  constructor factorization'

  field multiset  : Multiset
        allPrime  : All IsPrime (msKeys multiset)
        productEq : a ~ msValue multiset

  M = multiset

  aList : AssocList
  aList = msAL M

  ftPairs : List (C × ℕ)
  ftPairs = pairs aList        -- список пар  (p , кратность p в a)

  primes : List C
  primes = map proj₁ ftPairs    -- простые элементы разложения

  asdPrimes : List Associated
  asdPrimes = map asd primes

  field noRepetInAsdPrimes : ¬ AsdL.AnyRepeats asdPrimes
  ...
```

Операция `multiset` и аксиома `allPrime` объяснялись в предыдущих разделах.

`v = msValue multiset` есть значение многомножества `multiset` в мультипликативном моноиде `IR` (произведение степеней пар, см. Раздел 3).

Аксиома `productEq` означает, что `a` и `v` делят друг друга. В данном случае — отличаются обратимым множителем.

Список `asdPrimes` получается отображением каждого простого `p` из списка разложения в элемент `(asd p)` моноида `Associated`.

Аксиома `noRepetInAsdPrimes` означает, что список `asdPrimes` не содержит повторов.

9.3. Перенесение разложения между кольцом и моноидом ненулевых элементов

В данной среде запрограммированы доказательства некоторых простых лемм. В частности, доказаны леммы перенесения разложения:

Agda

```
fromNZFactorization : ∀ {a} → (a≉0 : a ≉ 0#) →
                      NZPack1.Factorization (a , a≉0) →
                      Factorization a

toNZFactorization : ∀ {a} → (ft : Factorization a) →
                    NZPack1.Factorization (a , nonzero-a ft)
```

Первая функция отображает разложение элемента `(a , _)` в CC-моноиде `Nonzero` в разложение элемента `a` в кольце `IR`.

Вторая функция делает обратное отображение. Причём в самом выражении типа второй функции содержится вывод `(nonzero-a ft)` неравенства `a ≉ 0#`. Ибо определение разложения на простые в `IR` таково, что из существования такого разложения для `a` конструктивно выводится неравенство `a ≉ 0#`.

Конечно, эти функции содержат все необходимые машинно-проверяемые доказательства.

9.4. Кольцо с разложением на простые множители

В той же среде целостного кольца `IR`, в модуле `OverIntegralRing-5` определено понятие кольца с разложением на простые множители:

Agda

```
module NZPack1 = OfCancellativeCommMonoid-1 *upCMon-NZ *cancel-nz _|'?_

record FactorizationRing : Set where
      field
         ftMonoid-nz :  NZPack1.FactorizationMonoid
      ...
```

Здесь `*upCMon-NZ` есть коммутативный моноид для множества `Nonzero`.

`*cancel-nz` есть свидетельство для закона сокращения в моноиде `Nonzero` (простой вывод этого свойства из определения целостного кольца запрограммирован ранее).

`_|'?_` есть функция распознавания делимости в моноиде `Nonzero`, она запрограммирована через функцию `_|?_` для `IR`.

Сим объявляется, что кольцо с разложением на простые есть структура моноида с разложением на простые для моноида `Nonzero`.

Далее, из структуры `ftMonoid-nz` берётся операция `factor` разложения в CC-моноиде `Nonzero`, и она переименовывается в `factor-nz`. Теперь функция разложения ненулевого элемента на простые в кольце `IR` выражается как

Agda

```
factor : ∀ a → a ≢ 0# → Factorization a
factor a a≢0 = fromNZFactorization a≢0 (factor-nz (a , a≢0))
```

Таким образом определения структуры Factorization и понятия FactorizationRing (с операцией factor) полностью выражают классические понятия для целостного кольца: *разложение на простые множители* и *кольцо с разложением на простые множители*.

9.5. Понятие однозначности разложения в целостном кольце

Не все кольца с разложением на простые обладают однозначностью разложения.

Классическое понятие *факториального кольца* ([10], глава II, параграф 4) включает свойство однозначности разложения.

Из соображений удобства в библиотеке DoCon-A принят следующий подход:

- Понятие факториального кольца явно (в виде записи) не определяется.
- Определяется понятие однозначности разложения (в целостном кольце).
- Свойство однозначности разложения выводится из свойства `prime|split` расщепления делимости для целостного кольца (на самом деле эти свойства равносильны, а условие `prime|split` гораздо проще записывается).
- Если программа пользователя или какая-либо функция библиотеки нуждается в параметре — факториальном кольце, то в модуль подставляются два параметра: кольцо с разложением на простые и свидетельство `prime|split` для этого кольца.
- Модули `UniqueFtRing-0` и `UniqueFtRing-1` библиотеки выводят из свойства `prime|split` и структуры кольца с разложением на простые различные функции и утверждения, среди них — утверждение `factorizationIsUnique` об однозначности разложения.

Таким образом параметрические программные модули `UniqueFtRing-0` и `UniqueFtRing-1` представляют понятие факториального кольца и некоторые функции и леммы с ним связанные.

Определение свойства расщепления делимости (на простой элемент) для целостного кольца дано в модуле `OverIntegralRing-2`:

Agda

```
Prime|split : Set _
Prime|split = ∀ {p a b} → IsPrime p → p ∣ (a * b) → p ∣ a ⊎ p ∣ b
```

Понятие однозначности разложения в целостном кольце IR определяется в модуле `OverIntegralRing-3`:

Agda

```
FactorizationIsUnique : Set _
FactorizationIsUnique =
 ∀ {a b} → a ≈ b → (ft : Factorization a) → (ft' : Factorization b) →
                        (asdPairs ft) ~pset (asdPairs ft')
```

Здесь `asdPairs` есть список пар из разложения, в которых ключ отображён в моноид `Associated` классов взаимной делимости для `IR`. Отношение `~pset` означает равенство *множеств* пар над областью `Associated`, задаваемых полученными списками, причём ключи в парах сравниваются как элементы сетоида `Associated`.

Формальный вывод свойства `FactorizationIsUnique` из свойства `Prime|split` оказался технически сложным:

(*1*) Случаи `a ≈ 0#` или `b ≈ 0#` рассматриваются отдельно — в этих случаях разложения не существует. В случае ненулевых `a` и `b` применяется отображение `toNZ` в мультипликативный CC-моноид `Nonzero` ненулевых элементов в `IR`.

(*2*) Из свидетельства `prime|split` выводится свидетельство `NZ.prime|split` для моноида `Nonzero`.

(*3*) Для моноида `Nonzero` ранее из условия `NZ.prime|split` выведено следующее утверждение, которое назовём здесь «Предединственность»:

«Пусть простой элемент `p` делит `a`, `ft` есть любое разложение на простые для `a`. Тогда в списке пар `ft` есть пара `(q , e)`, в которой ключ `q` равен `p` с точностью до обратимого множителя.»

В этой формулировке все понятия определены в смысле CC-моноида `Nonzero`.

(*4*) Для целостного кольца `IR` доказывается подобная лемма «Предединственность» Она доказывается перенесением утверждения (3) посредством отображения `toNZ : IR → Nonzero`.

(*5*) Из леммы (4) выводится очень близкое по смыслу утверждение — свойство однозначности `FactorizationIsUnique`.

9.6. Случай евклидова кольца

К любому евклидову кольцу `E` применим алгоритм расширенного НОД: для любых `a` и `b` выдаются `u` и `v` такие, что `a*u + b*v = НОД(a, b)`.

В модуле `OfEuclideanRing` библиотеки запрограммирован этот алгоритм `extGCD` и доказано его свойство расширенного наибольшего общего делителя. Также в этом модуле свойство `prime|split` выведено из наличия функции расширенного НОД. В свою очередь, из свойства `prime|split` ранее выведена однозначность разложения на простые в кольце `E` — если `E` есть кольцо с разложением на простые.

Чтобы программа воспользовалась однозначностью разложения на простые в евклидовом кольце `E` нужно эту однозначность (`factorizationIsUnique`) взять из параметрического модуля

Agda

```
module OfEuclideanFactorizationRing
     (upIR : UpIntegralRing _ _)
     (_|?_ : let Mg = UpIntegralRing.*magma upIR
          in Decidable₂ (Magma._|_ Mg)
     )
     (open OverIntegralRing-0 upIR _|?_ using (EuclideanRing))
     (open OverIntegralRing-5 upIR _|?_ using (FactorizationRing))
     (E    : EuclideanRing)
     (ftRing : FactorizationRing)
  where
  open OfEuclideanRing upIR _|?_ E  using (prime|split)
  open UniqueFtRing-1  upIR _|?_ ftRing prime|split
                         using (factorizationIsUnique)
```

То есть надо подставить в этот модуль следующие параметры:

- целостное кольцо `IR`,
- евклидово кольцо `E` (на основе кольца `IR`),
- кольцо с разложением на простые `ftRing` (на основе кольца `IR`).

10. Структура области с однозначным разложением на простые множители для натуральных и целых чисел

Для задания структуры моноида с разложением на простые множители для ненулевых натуральных чисел необходимо запрограммировать алгоритм разложения на простые для этого моноида. В библиотеке воплощён известный издревле способ решета, он сопровождается формальным доказательством.

Различные способы более высокой производительности потребуют более сложных доказательств, это исследование отложено на будущее.

Разложение на простые для (евклидова) кольца целых чисел запрограммировано через разложение на простые для ненулевых натуральных чисел.

Однозначность разложения для целых чисел выведена как частный случай построения «евклидово кольцо с разложением на простые ⟶ однозначность разложения»:

Agda

```
open OfEuclideanRing Int1.upIntegralRing _|?_ Int1.euclideanRing
   public using (prime|split)

open UniqueFtRing-1 Int1.upIntegralRing _|?_ factorizationRing prime|split
   public using (factorizationIsUnique)
```

Однозначность разложения для натуральных чисел выведена следующим образом:

- Свойство `prime|split` для $\mathbb{N}\backslash 0$ выведено из свойства `prime|split` для $\mathbb{Z}$.
- Однозначность разложения для $\mathbb{N}\backslash 0$ следует как в общем случае CC-моноида с разложением на множители и обладающего свойством `prime|split`.

11. Способ решета и принцип Маркова

Этот раздел повторяет часть тезисов [**16**].

Приводимые построения относятся к натуральным числам, представленным в унарной системе (Раздел 4.1).

Также используется то, что язык Agda основан на «ленивом» способе выполнения программы.

При программировании доказательства для способа решета распознавания простоты натурального числа открылось любопытное явление, связанное с требованиями конструктивной математики. Функция

Agda

```
nextPrime : ∀ p → IsPrime p →
         (∃ \ {q} → p < q × IsPrime q × IsFirstPrimeAfter q p)
```

должна найти первое простое число, большее простого числа `p`. Программа `nextPrime` перебирает числа `n = p+1, p+2, ...`, проверяя простоту каждого из них. Она выдаёт результат `n`, когда `n` окажется простым (оптимизации поиска в этой статье не рассматриваются).

Язык Agda требует конструктивного доказательства. В конструктивной математике всякий объект существует как алгоритм его построения, причём должно быть дано доказательство завершаемости этого алгоритма. В данном примере проверяльщик типов (type checker) системы Agda не видит такого доказательства и не принимает программу. Доказать завершаемость здесь можно, предъявив выражение (частный случай алгоритма) для построения числа `b > p`, с доказательством простоты `b`. Тогда добавим в цикл поиска значение `cnt` счётчика, которое начинается с `cnt = b` и каждый раз уменьшается на `1`, и так, чтобы при `cnt = 0` цикл заканчивался. Такая программа убеждает систему Agda в завершаемости алгоритма.

Известен простой способ построения такого `b`:

(*1*) `a` = *произведение всех простых от 2 до* `p` *включительно*
(*2*) `b` = *первый делитель для* `a+1`*, больший единицы, найденный, например, перебором.*

Нетрудно написать на языке Agda доказательства утверждений `p < b, IsPrime b`. Из этого всего получается простая доказательная программа для функции `nextPrime`.

Однако оказывается, что во время исполнения она уже для `p = 31` требует огромного времени для нахождения `q`. Причина состоит в том, что в цикле присутствует сравнение `cnt` с нулём по приводимому выше алгоритму: «произведение предыдущих простых ... искать первый делитель ... ». А это вычисление в среднем дорого стоит, когда `p > 30`.

Пробуется следующий выход из положения. Известна теорема Бертрана–Чебышёва о том, что между `p` и `2*p` есть простое число. Заменим границу перебора на `b2 = 2*p`.

При исполнении этот цикл работал бы быстро. Но проверяльщик типов теперь требует формального конструктивного доказательства теоремы Бертрана – Чебышёва. А это доказательство значительно труднее записать формально, чем доказательство свойства границы `b`.

Однако существует способ обойтись лёгким доказательством. Надо применить алгоритм поиска с двумя циклами:

(*1′*) сначала выполнить цикл поиска до b2 = 2*p;
(*2′*) если простое найдено, то выдать результат,
(*3′*) иначе выполнить поиск от p+1 до b.

Тогда

(*а*) Agda видит завершаемость программы,
(*б*) при исполнении сравнение счётчика в цикле (*1′*) выполняется быстро,

(*в*) а до исполнения цикла (*3′*) очередь не дойдёт.

11.1. Приём недостижимой границы

В другой задаче может оказаться, что верхней оценки с дешёвым сравнением счётчика для неё не известно. Ульф Норелл подсказал (в сообщении по электронной почте), что и для такого случая существует удовлетворительный способ программирования на языке Agda:

(*1″*) цикл выполнять до некоторого «недостижимого» (unfeasible) числа `B`, например, `B` = 2^{1000};

(*2″*) если объект не найден, выполнять цикл до границы `b`.

Тогда опять:

(*а*) дано доказательство завершаемости программы,

(*б*) до выполнения цикла (*2″*) программа не дойдёт,

(*в*) сравнение `cnt` $\stackrel{?}{=}$ `0` в цикле (*1″*) выполняется много быстрее, чем в программе `nextPrime`.

Эта быстрота достигается за счёт «ленивого» вычисления очередного сравнения (2^{1000} `- n` $\stackrel{?}{=}$ `0`) для `n = p+1, p+2, ...`

Объясним, почему это так. Для начального значения `n = p+1` в левой части сравнения через `n` шагов появляется (согласно алгоритму для операции `_-_`) значение `((suc d) - 0)`, которое вычисляется в `(suc d)`. Здесь `d` можно считать символом, который при необходимости вычислился бы до очень большого числа, но который дальше не вычисляется, так как по определению сравнения `_`$\stackrel{?}{=}$`_` исполнитель сразу полагает этот итог значением «нет» `(no _)`. Что касается сравнения счётчика для последующих `n`, то при разумном составлении программы оно будет делаться за один шаг.

11.2. О принципе Маркова

Принцип Маркова в конструктивной математике означает допустимость доказательства от противного для завершаемости алгоритма. А. А. Марков считал такой подход интуитивно ясным и отмечал его отличие от правил западной школы интуиционизма [**2**]. Вообще, этот предмет важен, так как касается вопроса об охвате области конструктивных сущностей и области применимости доказательного программирования в той или иной системе, и потому, что область доказательно завершающихся по Маркову алгоритмов, скорее всего, шире, чем область доказательно завершающихся алгоритмов в смысле западной школы интуиционизма.

Для примера посмотрим, как могло бы выглядеть доказательство от противного для завершаемости алгоритма `nextPrime`.

Заметим, что в этой версии сравнение текущего `n` с границей `b` оказывается ненужным. Назовём эту версию алгоритма `nextPrime-M`.

«Цикл `nextPrime-M` перебирает числа `n = p+1, p+2, ...`, и остановится только по условию `IsPrime n`. Предположим, что этот цикл не завершается. Из этого доказывается по индукции, что всякое `n` большее `p` не является простым, в частности таково `n = b`. Это противоречит выводу о том, что `p < b` и `IsPrime b`. Поэтому алгоритм `nextPrime` завершается. »

На языке Agda такое рассуждение не выразимо, и алгоритм `nextPrime-M` задать нельзя.

Как показано выше, вместо него задаётся алгоритм `nextPrime`, с «недостижимой» границей и двумя циклами. Видно, что этот приём ведёт к весьма широкой области алгоритмов, для которых принцип Маркова заменим, и не только в части формального доказательства, но и в части стоимости вычисления при исполнении.

12. Реализация библиотеки

Все функции, описанные в этой статье и в статьях [**1**, **6**], реализованы в выпуске 0.04.1 библиотеки DoCon-A [**17**].

13. Выводы

13.1. Выразительность языка

По текущему опыту, язык Agda ныне представляется автору наилучшим средством программирования математических вычислений (с учётом пристрастия автора к чисто функциональным языкам с «ленивым» способом вычисления).

В частности это подтверждается описанным в данной статье примером выражения на языке Agda понятия разложения на простые множители в областях различного вида, с доказательными программами для соответствующих алгоритмов и утверждений, и доказательной программой для способа решета разложения на простые.

Требуются некоторые усиления в реализации языка и расширение библиотеки.

13.2. Затраты проверки типов

Необходимо улучшить реализацию языка в отношении затрат на проверку типов (очевидно, что это выполнимо). Так, для компилятора Agda-2.5.2, ghc-7.10.2 и персонального компьютера частоты 3 GHz проверка типов (в частности — всех доказательств) библиотеки DoCon-A 0.04.1 требует минимума в 7 Гбайт ОЗУ и занимает 64 минуты.

13.3. Трудоёмкость составления формальных доказательств

Эта сложная проблема является главной для данной отрасли знания. Автору пока не понятно, является ли она чисто технической или ещё и научной.

Считается, что применение некой развитой программы-доказывателя могло бы облегчить составление доказательств. Но, например, для библиотеки DoCon-A-0.04.1 это облегчение едва ли будет существенным. Ибо существующие доказыватели способны самостоятельно заполнить только те части доказательств в программах DoCon-A-0.04.1, которые довольно просто составляются вручную, в то время как остальные части — трудоёмки.

Пример задачи на формальное доказательство: доказать ассоциативность и коммутативность действия сложения натуральных чисел в двоичном представлении. Насколько уменьшит трудоёмкость интерактивное применение доказывателя в данном примере в сравнении с ручным составлением? Автору представляется, что разница будет несущественной — если только не будет разработан доказыватель существенно более мощный, чем ныне имеются.

В целом, суть препятствия состоит в следующем.

В строгих (по меркам человека) конструктивных доказательствах (например, приводимых в учебниках) имеется много пропущенных подробностей, которые считаются очевидными и являются таковыми. Но многие из этих пробелов ни один из существующих автоматических доказывателей не способен самостоятельно заполнить формальным доказательством.

Благодарности. Автор благодарен А. П. Немытых за полезные замечания по изложению.

Список литературы

[1] С. Д. Мешвелиани. «О зависимых типах и интуиционизме в программировании математики», *Программные системы: теория и приложения*, **5**:3(21) (2014), с. 27–50, URL: http://psta.psiras.ru/read/psta2014_3_27-50.pdf ↑ 3,4,7,8,17,41

[2] А. А. Марков. «О конструктивной математике», *Проблемы конструктивного направления в математике*, Сборник работ. Т. 2: *Конструктивный математический анализ*, Тр. МИАН СССР, т. **67**, Изд-во АН СССР, М.–Л., 1962, с. 8–14. ↑ 3,40

[3] P. Martin-Löef. *Intuitionistic Type Theory*, Bibliopolis, 1984. ↑ 3

[4] U. Norell, J. Chapman. *Dependently Typed Programming in Agda*, URL: http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf ↑ 3,4,8,21

[5] *Agda. A dependently typed functional programming language and its system*, Homepage of Agda, URL: http://wiki.portal.chalmers.se/agda/pmwiki.php ↑ 3,10

[6] С. Д. Мешвелиани. «Программирование основ вычислительной алгебры на языке с зависимыми типами», *Программные системы: теория и приложения*, **6**:4(27) (2015), с. 313–340, URL: http://psta.psiras.ru/read/psta2015_4_313-340.pdf ↑ 4,8,11,41

[7] А. И. Кострикин. *Введение в алгебру. Основы алгебры*, Наука. Физматлит, М., 1994. ↑ 4

[8] Дж. Коллинз, Р. Лоос. *Компьютерная алгебра. Символьные и алгебраические вычисления*, ред. Б. Бухбергер, Мир, М., 1986. ↑ 4

[9] Дж. Давенпорт, И. Сирэ, Э. Турнье. *Компьютерная алгебра*, Мир, М., 1991. ↑ 4

[10] С. Ленг. *Алгебра*, Мир, М., 1968. ↑ 4,25,35

[11] N. Jacobson. *Basic Algebra I*, W. H. Freeman and Company, New York, 1985. ↑ 4,5,25

[12] S. D. Mechveliani. «Computer algebra with Haskell: applying functional-categorial-'lazy' programming», International Workshop CAAP-2001 (Dubna, Russia, 2001), с. 203–211, URL: http://compalg.jinr.ru/Confs/CAAP_2001/Final/proceedings/proceed.pdf ↑ 4

[13] S. Watt, et al.. *Aldor Compiler User Guide*, IBM Thomas J. Watson Research Center, URL: http://www.aldor.org ↑ 6

[14] *The Coq Proof Assistant*, Homepage of the Coq system, URL: http://coq.inria.fr ↑ 6

[15] A. Mahboubi. "The Rooster and the Butterflies", *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, vol. **7961**, Springer-Verlag, Berlin–Heidelberg, 2013, pp. 1–18. ↑ 6

[16] S. Meshveliani. “Provable programming of algebra: particular points, examples”, *Polynomial Computer Algebra 2016*, Collection of theses (Eulers’ International Mathematical Institute, Sankt-Petersburg, 2016), VVM Publishing, SPb., pp. 61–64. ↑ [38]

[17] С. Д. Мешвелиани. *DoCon-A, библиотека доказательных программ для построителя алгебраических областей, выпуск 0.04.1. Руководство (на английском языке) и исходный код* (Переславль-Залесский, 2016), URL: http://www.botik.ru/pub/local/Mechveliani/docon-A/ ↑ [41]

Рекомендовал к публикации *д.ф.-м.н. Н.Н. Непейвода*

Пример ссылки на эту публикацию:

С. Д. Мешвелиани. «Программирование вычислительной алгебры на основе конструктивной математики. Области с разложением на простые множители», *Программные системы: теория и приложения*, 2017, **8**:1(32), с. 3–46. URL: http://psta.psiras.ru/read/psta2017_1_3-46.pdf

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

Sergei Meshveliani. *Programming computer algebra with basing on constructive mathematics. Domains with factorization.*

Abstract. The paper continues prevoius author publications on a constructive mathematics approach in computational algebra using a language with dependent types.

A constructive expression for the notion of a *domain with factorization* to primes for a *monoid* and for a *ring* possessing certain additionnal properties is obtained. A way to achieve constructed machine-checked proofs for theorems that relate the factorization notion for domains of different kinds is discussed.

All the described methods and proofs are completely implemented as a working program written in the Agda functional language. (*In Russian*).

Key words and phrases: constructive mathematics, computer algebra, dependent types, functional programming, Agda.

References

[1] S. D. Meshveliani. "On dependent types and intuitionism in programming mathematics", *Programmnyye sistemy: teoriya i prilozheniya*, **5**:3(21) (2014), pp. 27–50 (in Russian), URL: http://psta.psiras.ru/read/psta2014_3_27-50.pdf

[2] A. A. Markov. "On constructive mathematics", *Problemy konstruktivnogo napravleniya v matematike*, Sbornik rabot. V. 2: *Konstruktivnyy matematicheskiy analiz*, Tr. MIAN SSSR, vol. **67**, Izd-vo AN SSSR, M.–L., 1962, pp. 8–14 (in Russian).

[3] P. Martin-Löef. *Intuitionistic Type Theory*, Bibliopolis, 1984.

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

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

[6] S. D. Meshveliani. "Programming basic computer algebra in a language with dependent types", *Programmnyye sistemy: teoriya i prilozheniya*, **6**:4(27) (2015), pp. 313–340 (in Russian), URL: http://psta.psiras.ru/read/psta2015_4_313-340.pdf

[7] A. I. Kostrikin. *Introduction to Algebra. Fundamentals of algebra*, Nauka. Fizmatlit, M., 1994 (in Russian).

[8] R. Albrecht, B. Bukhberger, G. E. Collins, R. Loos (eds.), *Computer Algebra: Symbolic and Algebraic Computation*, Computing Supplementa, vol. **4**, 2nd ed., Springer, 1983, 283 p.

[9] J. H. Davenport, Y. Siret, E. Tournier. *Computer Algebra*, Seconde edition, Academic Press, 1993, 298 p.

[10] S. Lang, *Algebra*, Graduate Texts in Mathematics, vol. **211**, 3rd edition, Springer, M., 1993.

[11] N. Jacobson. *Basic Algebra I*, W. H. Freeman and Company, New York, 1985.

© S. D. Meshveliani, 2017
© Ailamazyan Program Systems Institute of RAS, 2017
© Program systems: Theory and Applications, 2017

[12] S. D. Mechveliani. “Computer algebra with Haskell: applying functional-categorial-’lazy’ programming”, International Workshop CAAP-2001 (Dubna, Russia, 2001), pp. 203–211, URL: http://compalg.jinr.ru/Confs/CAAP_2001/Final/proceedings/proceed.pdf

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

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

[15] A. Mahboubi. “The Rooster and the Butterflies”, *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, vol. **7961**, Springer-Verlag, Berlin–Heidelberg, 2013, pp. 1–18.

[16] S. Meshveliani. “Provable programming of algebra: particular points, examples”, *Polynomial Computer Algebra 2016*, Collection of theses (Eulers’ International Mathematical Institute, Sankt-Petersburg, 2016), VVM Publishing, SPb., pp. 61–64.

[17] S. D. Meshveliani. *DoCon-A a library for a provable algebraic domain constructor, release 0.04.1. Manual (in English) and source program* (Pereslavl-Zalessky, Russia, 2016), URL: http://www.botik.ru/pub/local/Mechveliani/docon-A/

Sample citation of this publication:

Sergei Meshveliani. “Programming computer algebra with basing on constructive mathematics. Domains with factorization”, *Program systems: Theory and applications*, 2017, **8**:1(32), pp. 3–46. (*In Russian*).

URL: http://psta.psiras.ru/read/psta2017_1_3-46.pdf