С. Д. Мешвелиани

Построение доказательных программ арифметики натуральных чисел в двоичном представлении

Аннотация. Поддержка *зависимых типов* в функциональном языке программирования **Agda** создаёт возможность включать в программу машинно-проверяемые доказательства. Рассмотрена задача доказательного включения алгоритмов арифметических действий над натуральными числами в двоичном представлении.

Построена библиотека доказательных программ алгоритмов обычных письменных вычислений, действующих над списками двоичных разрядов чисел. Она содержит машинно-проверяемые доказательства необходимых свойств применённых алгоритмов, исправляет и существенно дополняет часть **Bin** стандартной библиотеки **lib-0.16** языка **Agda**.

Ключевые слова и фразы: компьютерная алгебра, арифметика двоичных кодов, доказательное программирование, язык **Agda**.

1. Введение

В программах на алгоритмическом языке обычно описываются *значения* и *типы* (множества значений). *Зависимые типы* суть те типы, которые зависят от значений [1], [6]. Например, тип всех списков длины n зависит от числа n .

Наличие зависимых типов даёт возможность включать в программу машинно-проверяемые доказательства в виде обычного программного кода функций. Это значит, что помимо алгоритма вычисления программа может содержать математические определения и утверждения, и эти последние могут относиться к алгоритму. Такие определения и утверждения представляются в виде объявлений типов с использованием соответствия Карри-Ховарда ([6], [7]). В этом соответствии

Исследование выполнено в рамках госзадания ФАНО России № г/р АААА-А16-116021760039-0.

© С. Д. Мешвелиани, 2018

© Институт программных систем имени А. К. Айламазяна РАН, 2018

© Программные системы: теория и приложения (дизайн), 2018

10.25209/2079-3316-2018-9-4-509-560

- математическому (логическому) высказыванию соответствует тип,
- доказательству высказывания соответствует любое выражение, принадлежащее данному типу,
- недоказуемое высказывание соответствует пустому типу $\perp$ с пустым множеством значений,
- импликация соответствует типу функций между соответствующими типами,
- конъюнкции соответствует прямое произведение $_ \times _$ типов,
- дизъюнкции соответствует сумма $_ \oplus _$ типов,
- отрицание высказывания соответствует типу всех функций, отображающих данный тип в пустой тип.

Задание таких типов и действия с ними адекватно поддерживаются например в изначально более ориентированном на программирование доказательств, чем на вычисления, языке **Gallina** [3]. Далее будет рассмотрен пригодный для вычислений язык программирования **Agda**¹, по сути расширяющий **Haskell** [2] конструкцией зависимых типов. Он также отличается чистой функциональностью и «ленивой» моделью вычислений.

Так, в доказательствах в **Agda**-программах применение леммы выражается в виде вызова функции, а доказательство по индукции выражается в виде рекурсии.

Конструктивные вычисления и доказательства с использованием закона *исключённого третьего* возможны только в случае алгоритмической разрешимости отношения. В частности, вычисления и доказательства вида «Если $x < y$, то способ_1, иначе способ_2» предполагают определённую функцию сравнения «<» в рассматриваемой области.

Следующий иллюстративный пример вводит неподготовленного читателя в доказательное программирование.

1.1. Пример доказательства на языке Agda

ПРИМЕР 1. *Формализовать и доказать утверждение*

- (1) *«для любого натурального числа n
выполнено свойство $n + 1 = 1 + n$ ».*

¹<http://wiki.portal.chalmers.se/agda/pmwiki.php>

Натуральные числа представлены в унарной записи, с конструктором `suc` для построения следующего числа. Например, число 2 представимо в виде `suc (suc 0)`. Функция сложения `+_` определена в виде рекурсивной программы, использующей сопоставление с образцом:

```

1  _+_ : ℕ → ℕ → ℕ      -- S → T  есть тип функций из типа S в тип T
2  0      + n = n
3  (suc m) + n = suc (m + n)

```

Для любого типа T выражение « $x : T$ » означает объявление « x имеет тип T ». Целевое высказывание (1) в нашем примере выражается в виде типа языка `Agda` следующим образом:

```

1  (n : ℕ) → n + 1 ≡ 1 + n      -- назовём этот тип F

```

точнее — в виде семейства типов, зависящего от параметра n . Такой параметр называется *индексом* для типа. Любая функция `lemma`, имеющая тип F , объявляется как

```

1  lemma : (n : ℕ) → n + 1 ≡ 1 + n

```

Для любого утверждения любое выражение, принадлежащее типу, соответствующему этому утверждению в смысле Карри-Ховарда, называется *свидетельством* и конструктивным доказательством для этого утверждения. Так, в нашем примере целью является построить выражение — тело функции, имеющей тип F .

Тип $n + 1 \equiv 1 + n$ есть тип свидетельств этого равенства для всех n . Тип F построен из типов $(n : \mathbb{N})$ и $(n + 1 \equiv 1 + n)$ применением конструктора « $\rightarrow$ ». Он описывает множество всех отображений, отображающих всякое натуральное число n в свидетельство для равенства $n + 1 \equiv 1 + n$.

Для функции f типа F и значения $n : \mathbb{N}$ вызов $(f\ n)$ должен согласно смыслу функции в языке `Agda` построить такое свидетельство. Любая такая функция даёт конструктивное доказательство утверждения (1). Если компилятору не предъявлено никакой функции типа F , то он считает, что доказательство отсутствует. Рассмотрим пример доказательства утверждения (1):

```

1  lemma : (n : ℕ) → n + 1 ≡ 1 + n
2  lemma 0      = refl
3  lemma (suc n) = cong suc (lemma n)

```

Здесь тип $\mathbb{N}$, конструктор `suc` («следующий»), отношение $\equiv$ равенства, конструктор `refl` и функция `cong` для правил вывода определены в стандартной библиотеке. Доказательство построено индукцией по построению значения n и представлено в виде тела функции `lemma`. Единицу компилятор заменяет на выражение $(\text{suc } 0)$. Первое предположение функции представляет доказательство основания индукции. Соответственно, в правой части стоит доказательство утверждения $0 + (\text{suc } 0) \equiv (\text{suc } 0) + 0$.

Символьные вычисления (без придания значений переменным) выражения по программам, находящимся в поле видимости компилятора, называются *нормализацией*. Компилятор применяет нормализацию автоматически. Например, выражение $(\text{suc } 0) + n$ в нашем случае нормализуется к выражению `suc n`. Также и тип $(\text{suc } 0) + n \equiv \text{suc } n$ нормализуется к типу `suc n  $\equiv$  suc n`.

Конструктор `refl` в первом предположении функции `lemma` означает, что обе части равенства

$$0 + (\text{suc } 0) \equiv (\text{suc } 0) + 0$$

нормализуются по программе для функции $+$, и нормальные формы сравниваются синтаксически. Функция $+$ определена выше так, что эта нормализация даёт равенство $\text{suc } 0 \equiv \text{suc } 0$. Его доказательством является конструктор `refl`.

Функция `lemma` для любого $k : \mathbb{N}$ должна в соответствии с её типом построить доказательство равенства

$$k + (\text{suc } 0) \equiv (\text{suc } 0) + k.$$

Она определена рекурсивно. Число k может иметь в унарной системе вид 0 или вид $(\text{suc } n)$. Функция `lemma` задана через сопоставление с образцом. Поэтому в левой части первого предположения естественно поставить образец `(lemma 0)`, а в левой части второго предположения естественно поставить образец `(lemma (suc n))`. Тогда в доказательстве будет разобран полный набор случаев.

Случай `(lemma 0)` соответствует основанию индукции.

Случай `(lemma (suc n))` представляет шаг индукции ($k = \text{suc } n$). Так как для любого k значение `(lemma k)` объявлено имеющим тип

$$k + (\text{suc } 0) \equiv (\text{suc } 0) + k,$$

то для любого n значение `(lemma (suc n))` должно иметь тип

$$(\text{suc } n) + (\text{suc } 0) \equiv (\text{suc } 0) + (\text{suc } n)$$

(компилятор подставляет $(\text{suc } n)$ вместо k). Поэтому в правой части второго предположения функции `lemma` компилятор ожидает построение

доказательства равенства

$$(2) \quad (\text{suc } n) + (\text{suc } 0) \equiv (\text{suc } 0) + (\text{suc } n).$$

Левая часть этого равенства нормализуется к выражению

$$\text{suc } (n + (\text{suc } 0))$$

в силу второго предложения функции $+$. Правая часть нормализуется к выражению $\text{suc } (\text{suc } n)$. Это приводит равенство (2) к виду

$$(3) \quad \text{suc } (n + (\text{suc } 0)) \equiv \text{suc } (\text{suc } n).$$

Компилятор делает нормализацию и заменяет цель доказательства (2) на цель (3). То есть в правой части второго предложения программы `lemma` достаточно построить доказательство равенства (3). В правой части этого предложения стоит выражение `cong suc (lemma n)`. Это композиция вызовов функций `lemma` и `cong`.

Она равносильна выражению

```
1 let
2   p : n + (suc 0) ≡ (suc 0) + n
3   p = lemma n
4 in
5 cong suc p    -- функция cong имеет два аргумента
```

По рекурсии, p есть некоторое доказательство равенства

$$n + (\text{suc } 0) \equiv (\text{suc } 0) + n.$$

Функция `cong` представляет аксиому и правило вывода из всякого равенства $x \equiv y$ равенства $f x \equiv f y$. Иначе говоря, для любой функции f и её аргументов x и y утверждение `cong f q : (f x ≡ f y)` выполнено, как только $q : x \equiv y$. Поэтому вызов `(cong suc p)` строит из p доказательство равенства

$$\text{suc } (n + (\text{suc } 0)) \equiv \text{suc } ((\text{suc } 0) + n).$$

Правая его часть нормализуется к выражению $\text{suc } (\text{suc } n)$, доказывая (3). Для компилятора это то же самое, что доказательство равенства (2). Доказательство построил вызов `cong suc (lemma n)`.

Мы увидели, что тело функции `lemma` представляет индуктивное доказательство, в котором рекурсия представляет шаг в индуктивном рассуждении. Задачей компилятора является проверка таких доказательств. □

1.2. Другие общие замечания

Перечисленные черты системы типов дают возможность автоматической проверки различных свойств программ. В частности — её правильности, когда эта правильность выражена в виде типа и дан исходный текст (*код*) функции соответствующего типа.

Программный модуль включает объявления типов и исходный код функций. В доказательном программировании программный модуль описывает не только алгоритмы, но и некоторую связанную с ними формальную теорию, содержащую *определения и доказательства*, воспринимаемые компилятором. Например, алгоритм упорядочения списка сопровождается определением понятия упорядоченности списка и включает (или сопровождается) доказательство(-ом) того, что результат является упорядоченным списком, содержащим те же элементы, что и исходный список. Другой пример: функцию двоичного кодирования числа естественно сопровождать доказательством взаимной однозначности этого отображения. То есть программа может представлять изложение некоторой теории, как в научной статье или в книге. Но эта теория должна быть полностью формальной и проверяемой компилятором. Она может быть посвящена каким-то вычислительным методам, а может быть ещё посвящена свойствам каких-то отношений. Например, высказывание «если простое число делит произведение целых чисел, то оно делит по крайней мере один из сомножителей» выражает свойство отношения делимости, и оно доказывается в различных программах доказательного программирования.

При разумном составлении программы наличие в ней доказательств не влияет существенно на производительность исполняемого кода, хотя неизбежны дополнительные затраты на компиляцию.

О логических теоретических основах: общая философия построения доказательств, применяемая при построении библиотеки, описана, например, в статье [4] о конструктивной математике (за исключением того, что Агда не поддерживает некоторый особый принцип Маркова [4], расширенно толкующий доказательство завершаемости алгоритма).

Реализация языка *Agda* в части представления конструктивных доказательств опирается на интуиционистскую теорию типов [5], на понятие изоморфизма Карри-Ховарда [6], позволяющее представлять логические (математические) утверждения в виде зависимых типов, а доказательства этих утверждений — в виде алгоритмов (функций, программ).

Работы [5] и [6] нужны лишь тем читателям, которые имеют соответствующий углублённый академический интерес. Ни для разработки библиотеки `Binary`, ни для её понимания читателем, ни для её использования в программировании нет необходимости изучать эти источники, достаточно интуитивного представления, описанного выше, и рассмотрения нескольких примеров.

1.3. Цель и результаты разработки

Целью данной работы является построение таких *доказательных* (сертифицированных) программ на языке `Agda` для арифметики двоичных записей натуральных чисел, которые имеют порядок стоимости вычисления (в смысле количества действий над разрядами) такой же, как известные алгоритмы способом «в столбик». Более производительные алгоритмы пока не рассматривались (хотя, например, можно было бы составить необходимые формальные доказательства для способа Карацубы [8] для умножения).

Главным результатом разработки является *реализация библиотеки* `Binary`, которая включает в себя объявленные выше черты. Она испытана в системе `Agda-2.5.4`. Эта библиотека реализована в выпусках 3.2 и 4. Исходные программы этих выпусков доступны соответственно на `www` страницах <https://github.com/mechvel/Binary-3.2> и <https://github.com/mechvel/Binary-4>. Данная статья описывает идеологию этой разработки. Выпуск `Binary-4` является улучшением по сравнению с выпуском 3.2. Версия 4 основана на другом представлении данных и других алгоритмах, она содержит более простые алгоритмы и доказательства. *Данная статья посвящена выпуску 3.2.* Но многие её части общего характера относятся и к выпуску 4.

Арифметика в предлагаемой библиотеке реализована обычными алгоритмами «в столбик», действующими над списками двоичных разрядов. Построение соответствующих данной арифметике формальных и полных доказательств в программе оказывается не столь простым, как может показаться. Это видно из списка лемм, перечисленных в дальнейшем изложении, особенно это касается техники построения доказательства завершаемости алгоритма (Разделы 4.5, 6).

Полученная автором программа исправляет и существенно дополняет часть `Bin` стандартной библиотеки `lib-0.16` языка `Agda`. Эта программа может быть включена в указанную стандартную библиотеку.

В части `Bin` стандартной библиотеки недостаёт:

- доказательства изоморфизма арифметики унарных чисел и арифметики их двоичного представления (в частности, взаимной однозначности отображения кодирования `fromN`),
- действия вычитания, с доказательством его определяющих свойств,
- действия деления с остатком, с доказательством его определяющих свойств

и некоторых других действий и доказательств, которые будут названы в дальнейшем. Заметим, что первое из перечисленных доказательств является обоснованием самого замысла введения двоичного представления. Без него введение двоичного представления даже не имеет смысла.

Отсутствие такого формального обоснования привело к нехватке в стандартной библиотеке какой бы то ни было быстро исполняемой и притом доказательной арифметики. Именно: наивные алгоритмы для унарной арифметики сопровождаются всеми необходимыми доказательствами, но для их выполнения требуются неприемлемо большие ресурсы. А производительные алгоритмы унарной арифметики исполняются вызовом встроенного кода, и их свойства не подтверждены доказательствами. Наконец, арифметика двоичных записей имеет высокую производительность (в смысле порядка вычислительной стоимости) — но отсутствует формальное обоснование, то есть машинно-проверяемое доказательство изоморфизма с унарной арифметикой.

В основном предлагаемое в нашей библиотеке усиление состоит в том, что добавлены многие необходимые машинно-проверяемые доказательства. В частности добавлено доказательство того, что арифметика области $\mathbb{N}$ для унарной записи числа изоморфна арифметике области двоичных записей. Действия сложения и умножения из обеих арифметик запрограммированы на Агде в стандартной библиотеке. Это машинно-проверяемое доказательство делает осмысленным применение двоичной арифметики.

Конечно, неформальное доказательство такого изоморфизма известно давно и довольно широко. Но система *Agda* поддерживает полные формальные доказательства. Поэтому естественно ожидать от стандартной библиотеки наличия в ней соответствующего формального доказательства. Предложение состоит в том, чтобы добавить эти доказательства, разработанные автором.

Также в библиотеке *Binary* предложена реализация доказательных программ для следующих действий над двоично-представленными

числами: *вычитание, деление с остатком, НОД*. Их отсутствие существенно для стандартной библиотеки. Эти алгоритмы сопровождаются формальными доказательствами определяющих свойств для данных действий.

Более подробно все эти построения описаны ниже. Существенно, что доказательства вышеперечисленных основных свойств арифметики построены не для тех алгоритмов, для которых это легче сделать, а для алгоритмов способом в столбик, которые имеют порядок стоимости вычисления гораздо меньший, чем соответствующие действия над унарными записями. Подробности объяснены в дальнейших разделах.

Автор не первый, кто делает данную программную разработку для языка Agda. Ниже называются две другие работы. Одна из них содержит всю перечисленную функциональность, кроме действия НОД. Вторая — несколько неполна. Но обе представлены только в виде кода, отсутствуют статьи или документация, которые объясняли бы подход к проекту.

В данной же статье представлены объяснения. Что касается реализации алгоритмов (в части действий вычитания, деления с остатком, НОД) и формальных доказательств, то она разработана автором независимо. Наша статья содержит объяснения о том, как это сделано.

Об особенной части разработки: конструктивная математика требует доказательства завершаемости каждого используемого алгоритма (функции Агды). Для формального доказательства завершаемости в предлагаемой библиотеке часто используется счётчик — натуральное число в унарной записи. Этому есть альтернатива, но каждый подход связан с некоторым усложнением программы, свойственным данному подходу. Использование такого счётчика потребовало осторожности. Например, при неосторожном использовании счётчика для функции деления с остатком (Раздел 4.5) стоимость выполнения функции имела бы показательный порядок. Мы подробно рассматриваем этот тонкий аспект в Разделе 6, с разбором примера.

1.4. История разработки

Отметив вышеописанный пробел в стандартной библиотеке, автор решил запрограммировать недостающую часть. Это было любопытное упражнение в доказательном программировании. Оно потребовало гораздо больших усилий, чем автор ожидал.

Во время разработки автор узнал из писем, что уже несколько лет существует библиотека А. Алексеева (<https://github.com/Rotsor/BinDivMod>), которая заполняет этот пробел (возникает вопрос: почему в стандартную библиотеку не внесены соответствующие добавки?).

Как бы то ни было, этот маленький проект выполнен, и здесь даётся описание его идеологии и приводятся сопутствующие соображения и наблюдения о доказательном программировании.

Статья (даже без Приложения) занимает в три раза больший объём, чем было изначально намечено. Так получилось из-за добавления частей, которые вместе составили по сути некоторое руководство по программированию на языке *Agda*. Это сделано ради того, чтобы не заставлять читателя предварительно изучать такие руководства по книгам.

1.5. Предполагаемый круг читателей

Эта статья адресована тем, кто интересуется техникой доказательного программирования вычислений в математике на чисто функциональном языке. Эта техника показана на примере алгоритмов арифметики натуральных чисел в двоичном представлении. Существенно, что при этом испытываются практические возможности применения именно системы *Agda*.

Обширная библиотека формальных математических доказательств имеется для системы *Coq* [3], то есть возможности системы *Coq* уже проверены на большом круге задач.

Система *Agda* более молодая, она сильно отличается от системы *Coq*. Она предназначена для любителей чисто функционального программирования. Используется другая языковая парадигма.

В данной статье не разбираются различия этих двух систем. Мы описываем испытание системы *Agda* на определённом примере, и заодно получаем и описываем необходимое исправление стандартной библиотеки.

Эта система (язык + стандартная библиотека + реализация) требует практического испытания. Библиотека *Binary* представляет итог такого испытания на относительно несложном примере. Существует гораздо более сложный пример, в виде библиотеки программ вычислительной алгебры: S. D. Meshveliani, *DoCon-A* «A library of provable programs for

computer algebra»², Исходный текст программ и руководство, последний выпуск — 2018 года, Но здесь эта программа не обсуждается.

Какого рода испытания имеются в виду? Например, мы пишем «при разумном составлении программы наличие в ней машинно-проверяемых доказательств не понижает существенно производительности исполняемого кода». Но могут быть разные «подводные камни». Не окажется ли что доказательная программа вычитания имеет, например, квадратичный или даже показательный порядок стоимости выполнения тогда, как он должен быть линейным? Другой пример: не окажется ли запись определения НОД в виде зависимой записи такой громоздкой, что возникнет вопрос о применимости языка *Agda* к доказательному программированию алгебры? Не окажется ли, что для компиляции библиотеки требуется неестественно много ресурсов? И так далее.

В первом и последнем случаях надо будет выяснять причину: это

- (а) неудачное программирование *или*
- (б) недостаток реализации языка *или*
- (в) принципиальный недостаток применяемой алгоритмической теории типов.

Описанный здесь опыт подтверждает, что *в данном проекте* не возникает никаких подобных затруднений.

1.6. Содержание статьи

В разделе 2 даётся некоторое введение в язык *Agda*, кратко описываются различные представления для натуральных чисел и соответствующие способы программирования арифметики натуральных чисел.

В разделе 3 перечисляется, каких функций и доказательств не хватает в стандартной библиотеке в части арифметики двоичных записей, и называются соответствующие добавления из библиотеки *Binary*.

В разделе 4 излагается основное содержание разработки. Описываются функции арифметики двоичных записей, включая вычитание и деление с остатком, формализуются понятия с ними связанные, описывается построение доказательств их основных свойств. В частности, описывается построение доказательства изоморфизма (относительно

²<http://www.botik.ru/pub/local/Mechveliani/docon-A/>

сложения и умножения) арифметики натуральных чисел в унарном и двоичном представлениях.

В разделе 5 объясняется строение предлагаемой библиотеки в целом, перечисляются программные модули, из которых состоит библиотека, и для каждого модуля называются наиболее важные структуры и функции, которые он содержит.

В разделе 6 рассказывается об одном широко применяемом способе доказательства завершаемости функции в языке Agda и разбирается пример его применения к функции деления с остатком.

В разделе 7 даётся некоторый обзор других работ по данному предмету.

В заключении подытоживается представленный в статье опыт и указывается на возможный проект в смежной части библиотеки арифметики чисел.

Приложение I содержит пример программирования доказательства на языке Agda.

Приложение II содержит индекс имён типов и функций.

2. Некоторые вводные понятия о представлении чисел и языке Agda

О словоупотреблении: язык Agda мы также называем по-русски Агда, и склоняем это имя по падежам. Слово «библиотека» в данной статье означает только какую-то библиотеку программ. Слова «стандартная библиотека» означают стандартную библиотеку языка Agda. Арифметические законы ассоциативности, коммутативности и дистрибутивности называем соответственно законами *сочетательным*, *перестановочным* и *распределительным*. Слово «функция» обозначает всюду ниже программу (подпрограмму) на Агде (функциональном языке программирования).

О литературе: для читателя желательно предварительное знакомство с функциональным программированием вообще и с языком Agda (руководство [1] и статья [10]).

2.1. О синтаксисе языка Agda

Agda воспринимает символы в кодировке UTF-8. Это позволяет использовать в тексте программы математические обозначения $\mathbb{N}$, $\approx$, $\neq$, $\leq$, $\not\leq$, $\circ$, x^{-1} , $\neg$, $\oplus$, $\doteq$ и другие, включая символ конца доказательства ■.

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

Имена (идентификаторы) в Agda-программе разделяются (в основном) только пробелом. В частности имя оператора или переменной выделяется пробелами, непосредственно примыкая к объемлющим скобкам.

ПРИМЕР 2. Рассмотрим отрывок кода

```
1 m : ℕ
2 m = foo1
3 n = foo2
4 2*n≥m : 2 * n ≥ m -- объявление принадлежности типу
5 2*n≥m = f m n -- применение функции f
6 p = g 2*n≥m -- применение функции g к значению 2*n≥m из типа (2 * n ≥ m)
```

Здесь `2*n≥m` есть имя переменной. Оно обозначает значение в типе `2 * n ≥ m`. Символы «`*`», «`≥`» в выражении для этого типа обозначают соответственно оператор и конструктор типа, ибо они отделены пробелами.

Заметим, что имя `2*n≥m` является выразительным: в нём кратко отражен смысл этой переменной. □

2.2. Унарная запись натурального числа

В изначальном определении натуральное число представляется последовательностью одинаковых знаков — как ряд счётных палочек. Это есть запись числа в унарной системе. В стандартной библиотеке унарные записи натуральных чисел используют конструктор `suc` следующего за аргументом числа. Например, 0, 1, 2, и 3 представляются как 0, `suc 0`, `suc (suc 0)` и `suc (suc (suc 0))`. Все функции и данные в библиотеке `Binary` как и в стандартной библиотеке записаны на языке Agda).

Об использовании унарной арифметики: в некоторых функциях библиотеки `Binary` используется унарная арифметика натуральных чисел. При этом иногда она применяется не только в части доказательств, но и в части, которая вычисляется в исполняемом коде (примеры приведены в разделах 4.1, 6). Это делается так, чтобы её использование не увеличивало порядка стоимости вычисления целевой функции, даже в предположении, что унарная арифметика выполняется не встроенным,

а явным способом, как это показано в программах унарного сложения и умножения в разделах 2.3, 2.4.

2.3. Сопоставление с образцом. Стоимость вычисления

ПРИМЕР 3. Сложение в унарной записи есть соединение списков одинаковых знаков, задававшееся³ в стандартной библиотеке функцией

```

1  _+_  : N → N → N                                -- S → T  есть тип функций из типа S в тип T
2  0    + n = n
3  (suc m) + n = suc (m + n)

```

Первая строка объявляет тип функции. Выражение $S \rightarrow T \rightarrow U$ означает тип функций от двух аргументов, — типов S и T соответственно, — со значениями в типе U . Скобки расставляются так: $S \rightarrow (T \rightarrow U)$. То есть любая функция этого типа представляется как функция из типа S , каждое значение которой есть функция от аргумента из T .

Две другие строки составляют реализацию (тело) функции в виде предложений на языке сопоставления с образцом — как в языке Haskell. $\square$

Исполнитель программы пытается применить предложения в том порядке, в каком они расположены в программе. Все программы библиотеки `Binary` написаны на этом под-языке сопоставления с образцом (ещё применяются конструкции `where`, `let`, `case`, которые в основном сводятся к вызову функций, заданных через сопоставления с образцом).

Стоимость вычисления мы измеряем количеством действий сопоставления с образцом, сделанных в ходе выполнения программы. Например, при $m > 1$ вычисление суммы $m+n$ по приведённой выше программе требует $(1+2m)$ попыток сопоставления с образцом.

Такое понятие лишь приблизительно соответствует производительности программы исполняемой в действительности. Например, стоимость одного сопоставления с образцом зависит от размера образца, от того, как реализовано это сопоставление. Полностью адекватно формализовать понятие стоимости вычисления для машины сопоставления с образцом в объёме этой статьи не удастся. Ограничимся данным приблизительным понятием. Оно отвечает многочисленными

³В последних версиях стандартной библиотеки сложение задано встроенной функцией.

примерами практических вычислений, по крайней мере в части различия линейного, квадратичного, иного степенного порядка стоимости, или же показательного порядка. Например: при записи сортировки списка букв способом вставки и способом слияния, увидим что первая программа даёт квадратичный рост времени выполнения (в худшем случае) в зависимости от длины n списка, а вторая даёт рост порядка $O(n \cdot \log(n))$.

2.4. О реализации арифметики натуральных чисел в различных библиотеках

В изначальном определении (в виде счётных палочек) произведение $m * n$ натуральных чисел задано как повторное присоединение копии ряда m палочек: для каждого члена списка n к текущему итогу присоединяется список m . В старых версиях стандартной библиотеки⁴ это выражалось циклом:

```

1  _*_ : N → N → N
2  m * 0      = 0
3  m *(suc n) = m + (m * n)

```

Agda —

Для этих исходных понятий и алгоритмов нетрудно доказать некоторые их основные свойства: сочетательность и перестановочность сложения и умножения, распределительный закон, монотонность этих действий относительно отношения порядка и некоторые другие. Эти доказательства входят в программы стандартной библиотеки. Они являются формальными и проверяются компилятором автоматически.

Техника составления и применения таких формальных доказательств основана на инструменте *зависимых типов*, она объяснена в Примере 1.1, руководстве [1] и в статье автора [10].

На практике арифметические действия над числами в унарной записи неприемлемо затратны. Например, умножение «столбиком» чисел в двоичной записи имеет стоимость $O(l^2)$, где l есть количество разрядов в сомножителях. А соответствующее умножение в унарной записи стоит больше чем 2^l . Наиболее производительная арифметика натуральных чисел для современных (фон-неймановых двоичных) машин реализуется для представления числа в виде списка (или

⁴В последних версиях стандартной библиотеки умножение задано встроенной функцией.

массива) макроцифр, где макроцифра представлена машинным словом. Назовём такое представление *макроцифровым*. Вычислительная стоимость двоичной арифметики имеет тот же порядок, что и стоимость арифметики макроцифрового представления, но она отличается постоянным множителем. В библиотеке `Binary` реализованы действия сложения, вычитания, сравнения по упорядочению, умножения, деления с остатком, НОД. Для первых трёх и из них этот множитель ограничен сверху числом приблизительно равным E , где E число разрядов в машинном слове. Для умножения и деления с остатком ограничение есть E^2 , для НОД ограничение есть E^3 .

Для достижения удовлетворительной производительности современная версия стандартной библиотеки Агды выполняет арифметические действия над унарными записями (тип `ℕ`), вызывая встроенные функции из библиотеки `Glasgow Haskell`, которые обрабатывают числа в макроцифровом представлении. Выходит, что *доказательства в стандартной библиотеке относительно арифметики области $\mathbb{N}$ не относятся к тем алгоритмам, которые в действительности исполняют эту арифметику*. А свойства арифметики на $\mathbb{N}$ принимаются и используются как аксиомы, без доказательства.

Этот недостаток может быть исправлен почти полностью. Именно, арифметика макроцифрового представления может быть запрограммирована на языке `Agda` в виде явной обработки списков макроцифр, с необходимыми формальными доказательствами. Но для сохранения производительности понадобится некоторый небольшой набор действий над макроцифрами исполнять с помощью встроенных функций и некоторые свойства этих функций принять как аксиомы. Получится реализация достаточно производительной арифметики, в которой алгоритмы верифицированы на Агде вплоть до уровня области макроцифр. Поставленное выше слово «почти» относится к тому, что определяющие свойства некоторого небольшого набора действий в области макроцифр останутся недоказанными на Агде и будут приняты как аксиомы.

Если за макроцифру принять двоичный разряд, то получится арифметика двоичных записей. Здесь область цифр состоит из двух элементов, и арифметика цифр записывается на Агде без потери производительности и вместе с необходимыми доказательствами. Получается полностью доказательная арифметика натуральных чисел, имеющая производительность такого же порядка, как арифметика макроцифрового представления (но производительность меньше на вышеупомянутый постоянный множитель).

В современной стандартной библиотеке (`lib-0.16`) Агды реализация арифметики двоичных записей существенно неполна. Отсутствуют некоторые необходимые доказательства и некоторые необходимые операции (перечисленные в следующем разделе).

В этой статье описывается, каким образом построена библиотека `Binary`, исправляющая и дополняющая часть `Bin` стандартной библиотеки `lib-0.16` языка `Agda`.

2.5. Ещё о числах в стандартной библиотеке и их представлениях

Объясним более подробно положение арифметики чисел в стандартной библиотеке.

В стандартной библиотеке `lib-0.16` реализована арифметика для чисел следующих типов. Натуральные числа, целые числа (мало чем отличаются от натуральных в смысле структуры и алгоритмов), рациональные числа. Алгоритмы для всех этих арифметик опираются на арифметику натуральных чисел в унарном представлении. А при вычислениях эта унарная арифметика выполняется через встроенные функции.

В ранних версиях стандартной библиотеки арифметика унарных чисел выполнялась по наивным алгоритмам, записанным на Агде. Например, один шаг вычисления сложения выполнялся через сопоставление с образцом в правиле

```

-- Agda --
/ (suc m) + n = suc (m + n)

```

Именно для таких программ стандартная библиотека включает машинно-проверяемые доказательства (сертификаты) на языке `Agda` в части унарной арифметики. На практике такая арифметика неприемлемо медленна. Поэтому в последних версиях стандартной библиотеки арифметические действия над унарными числами выполняются не по алгоритмам, написанным на Агде, а через обращение к встроенным функциям, написанным на Си или на ассемблере. Это нетрудно увидеть, например, по скорости исполнения программ и по объявлению

```

-- Agda --
/ open import Agda.Builtin.Nat public using ( zero; suc; _+_; _*_ ) ...

```

в начале модуля `Data.Nat.Base.agda`. Наивные программы унарной арифметики остались лишь в старых версиях и в пособиях по программированию на Агде.

При этом доказательства свойств унарной арифметики, данные в стандартной библиотеке, более не являются сертификатами унарной арифметики, они относятся не к тем программам, по которым в действительности исполняются арифметические действия. Такой подход означает: «Арифметика унарных чисел — встроенная, её необходимые базовые свойства не доказаны на Агде, а постулированы для Агды. Но все остальные программы написаны на Агде и сертифицированы».

В стандартной библиотеке определяется тип `Bin`, который представляет натуральные числа в двоичной системе, но он нигде не применяется в указанной библиотеке.

Библиотека `Binary` исправляет стандартную библиотеку. С точки зрения наличия доказательств, двоичное представление натуральных чисел предпочтительнее. Ибо алгоритмы арифметики, не использующие встроенные функции, будут сертифицированы.

Но эта арифметика в десятки раз медленнее, чем встроенная арифметика для унарных чисел из `lib-0.16`.

Далее, имеет смысл разработать сертифицированные программы для натуральных чисел в представлении списком макроцифр, как описано выше. Эта арифметика будет промежуточной по скорости между двоичной и встроенной, и будет сертифицирована в том смысле, который объяснён выше в Разделе 2.4.

Если пользователь пренебрегает сертификацией арифметики в части натуральных чисел ради производительности кода, то он будет использовать тип $\mathbb{N}$ унарных чисел и его встроенную арифметику. А тот педант, кто желает опираться на сертифицированную арифметику натуральных чисел сможет пользоваться двоичным (тип `Bin`) или макроцифровым представлением (скажем, тип `NM`).

Второй подход имеет мало практического смысла, а имеет некоторый формально-теоретический смысл, смысл полноты доказанности. Всё-таки желательно иметь его реализацию.

Двоичная арифметика ещё имеет разные «точечные» применения. Например, в стандартную библиотеку можно было бы включить функцию быстрого возведения в степень в любом моноиде (двоичный

способ, с малым количеством умножений). Такое действие $x \hat{n}$ удобно программируется для натурального n , записанного в двоичном представлении.

Всё ли доказательное программирование встречается с такими неудобствами в части производительности, какие названы для арифметики натуральных чисел?

Нет. Натуральные числа — это исключение.

Например, пусть на Агде написано приложение: сертифицированные программы для быстрых алгоритмов вычислительной алгебры, скажем, для арифметики многочленов с рациональными коэффициентами. И оно опирается на библиотеку `lib-0.16` и применяет унарную арифметику натуральных чисел. И при этом на Агде доказывается перестановочность умножения многочленов. Насколько достоверно, что при исполнении умножения для конкретных многочленов f и g выполнится свойство $f * g \approx g * f$? Оно выполнится — если встроенная арифметика унарных натуральных чисел работает правильно.

То есть при использовании типа $\mathbb{N}$ (вместо `Bin` или `NB`) все Agda-сертификаты будут действительны — «по модулю» правильности арифметики унарных натуральных чисел (эта арифметика применяется почти всюду в различных библиотеках Agda-программ).

Натуральные числа являются исключением для доказательного программирования в вышеописанном смысле по следующей фундаментальной причине.

Для действий над макроцифрами Agda-доказательства их основных свойств возможны только если эти действия записаны как программы на Агде. Это противоречит требованию быстроты действий над цифрами. Например, если числа представлены в 16-ичной системе, то функция умножения цифр будет содержать 256 предложений языка Agda. Если же программируется система исчисления по основанию 2^{64} (как в современных библиотеках программ), то совсем уже не понятно, как представлять на Агде такие цифры, чтобы их арифметика была быстрой Agda-программой и чтобы эта программа была сертифицирована.

Заметим также, что библиотека `Binary` иногда использует унарную арифметику. Но это делается так, что это не увеличивает порядок стоимости вычисления, даже если унарная арифметика выполняется наивным способом по Агда-программе. Для надёжной проверки эффекта применения наивной унарной арифметики было бы хорошо иметь возможность глобально переключать встроенное выполнение

унарной арифметики на наивное, то есть иметь две версии стандартной библиотеки.

2.6. О составлении доказательства на языке Agda

Общие приёмы построения доказательств представлены в соответствии Карри-Ховарда, как это описано в начале Введения. Здесь же дадим более подробные объяснения.

Отступление об отношении равенства: ниже знак $\equiv$ означает в стандартной библиотеке синтаксическое (propositional) равенство — в отличие от определяемого (definitional) равенства $\approx$, задаваемого программистом по-своему для каждого рассматриваемого типа. Равенство $\equiv$ предопределено как общее для всех типов через правило рефлексивности $x \equiv x$. Например, для натурального числа n равенство $1 + n \equiv 1 + n$ имеет в качестве доказательства конструктор данных `refl`, ибо оно является подстановочным случаем правила рефлексии. Другой пример: равенство $0 + n \equiv n$ доказано конструктором `refl` или не доказано в зависимости от того, находится ли функция `_+_` для типа $\mathbb{N}$ в области видимости компилятора. Если она находится в поле видимости (импортирована), то компилятор вычисляет *нормализацию* выражения вызова этой функции по функциям из видимого окружения. Это приводит левую часть равенства к виду n , и опять, конструктор `refl` даёт доказательство.

Далее, для иллюстрации доказательного программирования разберём подробно часть доказательства некоторой леммы из нашей библиотеки (общее введение в программирование на Агде содержится в [1]).

ПРИМЕР 4. *Рассмотрим утверждение*

- (4) *«для любых двоичных записей a и b верно равенство $(\text{suc } a) + b \equiv \text{suc } (a + b)$ ».*

Это утверждение представляется типом

Agda —
 $\lambda (a\ b : \text{Bin}) \rightarrow (\text{suc } a) + b \equiv \text{suc } (a + b)$

Этот тип, который мы обозначим через $T(a, b)$, явно зависит от переменных a, b . Например, компилятор считает, что типы $T(a, b)$ и $T(a+b, b)$ разные. Любое значение из этого типа $T(a, b)$ представляет

доказательство данного утверждения. Поэтому, если какая-то функция (алгоритм) от a и b выдаёт значение этого типа, то эта функция представляет доказательство этого утверждения. Так что вышеприведённая лемма формулируется и доказывается формально в виде следующей программы:

```

1 suc-assoc : (a b : Bin) → (suc a) + b ≡ suc (a + b) -- формулировка утверждения
2 suc-assoc 0# b = cong suc (sym (0+ b))
3 suc-assoc a 0# = begin (suc a) + 0#    ≡( +0 (suc a) )
4                               suc a      ≡( cong suc (sym (+0 a)) )
5                               suc (a + 0#)
6
7
8 suc-assoc (bsA 1#) (bsB 1#) = ...

```

Здесь $0\#$ есть двоичная запись нуля, $(bsA\ 1\#)$ двоичная запись произвольного ненулевого числа. Проводя символьные вычисления с телом функции как с выражением от переменных a , b , компилятор убеждается, что эта функция возвращает итог нужного типа. И этот итог есть свидетельство выполнения утверждения для данных a и b .

Функция `suc-assoc` задана рекурсией по построению значений a и b . Основание рекурсии составляют случаи, когда первый или второй аргумент является нулём. Доказательство для этих случаев здесь приведено, а для случая ненулевых операндов $(bsA\ 1\#)$ $(bsB\ 1\#)$ (шаг индукции) заменено многоточием.

Разберём подробно только случай аргументов вида $0\#\ b$. Для него доказательство (свидетельство) есть значение `cong suc (sym (0+ b))`. Это значение должно иметь тип $(suc\ 0\#) + b \equiv suc\ (0\# + b)$.

Согласно программе для `_+_` для области `Bin` левая часть этого выражения типа упрощается, и тип нормализуется к выражению

$$suc\ b \equiv suc\ (0\# + b).$$

Лемма $0+$ утверждающая, что $0\# + x \equiv x$ для любого x в библиотеке доказана раньше. Применение $(0+ b)$ функции $0+$ к значению b представляет применение леммы к b . Отсюда $(0+ b)$ есть доказательство равенства $0\# + b \equiv b$.

Аксиома `sym` из стандартной библиотеки выражает симметричность равенства `_≡_`. Поэтому `sym (0+ b)` имеет тип $b \equiv 0\# + b$. По аксиоме `cong` из стандартной библиотеки применение любой функции f (подходящего типа) к частям любого равенства $l \equiv r$ выводит равенство $(f\ l) \equiv (f\ r)$. Теперь функция `(cong suc)`, применённая к

свидетельству равенства $b \equiv 0\# + b$, выдаёт свидетельство равенства

$$\text{suc } b \equiv \text{suc } (0\# + b).$$

Композиция нормализации по программе и всех этих применений даёт выражение `cong suc (sym (0+ b))` правой части, которое есть доказательство равенства

$$(\text{suc } 0\#) + b \equiv \text{suc } (0\# + b).$$

□

Как в своей полноте выглядят доказательства в библиотеке `Binary` показано в Приложении I. Там приведён отрывок кода, представляющий необходимые определения и доказательство утверждения о сложении двоичной записи с нулём.

Другие примеры машинно-проверяемых доказательств по индукции имеются в статье [10] и в исходных текстах библиотеки `Binary`.

О вычислении в доказательствах: некоторые доказательства для арифметики двоичных записей в библиотеке `Binary` ссылаются на алгоритм `toN`, который имеет показательный порядок вычислительной стоимости. Влияет ли это на производительность исполняемой программы?

В данном случае существенно не влияет, так как функция `toN` поставлена в правильных местах.

Отношение к вычислению в доказательном программировании таково, как в учебниках. В учебнике может описываться быстродействующий алгоритм и при этом приводиться доказательство его правильности, и это доказательство может ссылаться на алгоритм большой вычислительной стоимости. Что касается `Agda`-программ библиотеки `Binary`, функция `toN` поставлена в тех местах, где она *частично* вычисляется во время компиляции (во время проверки типов), но не во время исполнения. Там же, где она вычисляется при исполнении кода, это вычисление имеет малый порядок стоимости (как, например, при разрешении порядка на двоичных записях).

3. Что есть и чего не хватает в стандартной библиотеке для области `Bin`

3.1. Что есть в стандартной библиотеке для области `Bin`

Тип `Bit` содержит два значения: двоичные разряды `0b` и `1b`.

Конструктор типов `List` каждому типу `T` сопоставляет тип `(List T)` конечных списков элементов из `T`. Например, `(List Bit)` есть тип всех списков двоичных разрядов. Всякий тип вида `List T` имеет конструкторы данных `[]` (пустой список) и `_::_` (приставление элемента из `T` спереди к списку).

О синтаксисе: знак «`_`», приставленный слева или справа к имени оператора в объявлении типа оператора, означает, что этот оператор имеет аргумент, записываемый соответственно слева или справа от знака оператора. Например, оператор `—` конструктор `_::_` записывается с одним аргументом слева и одним аргументом справа.

Двоичная запись числа определена как тип данных

```

1 data Bin : Set where  0# : Bin
2                      _1# : (bs : List Bit) → Bin

```

Знак `#` есть часть *имен* этих двух конструкторов (пояснения о синтаксиса Агды даны в Разделе 2.1).

Младшие разряды ставятся в начало списка. `_1#` есть постфиксный конструктор, который соответствует приписыванию единицы как старшего разряда. Это представление даёт следующее соответствие:

```

1 0#           -- 0,
2 [] 1#        -- 1,
3 (0b :: []) 1# -- 2,
4 (1b :: []) 1# -- 3,
5 (0b :: 0b :: []) 1# -- 4,
6 (1b :: 0b :: []) 1# -- 5

```

и так далее.

Как видно, данное типа `Bin` есть почти то же, что данное типа `List Bit`. Они отличаются парой функций перекодировки: `toBits`, `fromBits`. Эти функции просты и имеют линейный порядок стоимости вычисления. Для примера запишем одну из них:

```

1 toBits : Bin → List Bit
2 toBits 0      = [ 0b ]      -- список (0b :: []) из одного элемента
3 toBits (bs 1#) = bs ++ [ 1b ] -- _++_ есть функция соединения списков

```

Заметим, что элементам `Bin` соответствуют при этом отображении не все списки двоичных разрядов, а только пустой список и ещё все те списки, которые заканчиваются единичным разрядом.

Эти списки в библиотеке `Binary` приняты за *второй канонический вид двоичной записи*. Условие принадлежности списка ко второму каноническому виду выражено функцией `HasLast1if` — «заканчивается на единицу, если не пуст».

ПРИМЕР 5. Число 0 во втором каноническом виде представлено списком `[]`, единица представлена списком `1b :: []`, число 5 представлено списком `1b :: 0b :: 1b :: []`.

□

Функция `toN : Bin → ℕ` реализует отображение из области `Bin` в область `ℕ` натуральных чисел в унарном представлении. Для каждой двоичной записи `x` (`toN x`) есть натуральное число, для которого `x` есть двоичный код. Функция `fromN : ℕ → Bin` выдаёт двоичную запись натурального числа.

Реализованы операции `_+_` and `_*_` для области `Bin`, операции `suc` («следующий»), `pred` («предыдущий»), `<-cmp` (сравнение вида «меньше-больше-равен»). Эти операции имеют естественный порядок стоимости. Например, действия `x + y` и `(<-cmp x y)` имеют стоимость

$$O(\max(|x|, |y|)),$$

где функция `|_|` выдаёт число разрядов в двоичной записи.

Стандартная библиотека реализует структуру разрешимого строгого линейного порядка (`StrictTotalOrder`) на области `Bin` относительно упорядочения `<-cmp`.

3.2. Чего не хватает в стандартной библиотеке для области `Bin`

Не хватает следующего.

- Доказательства взаимной однозначности отображения `toN` и того, что `toN` взаимно-обратно с `fromN`.
- Доказательства того, что `toN` и `fromN` сохраняют действия `_+_` и `_*_`.
- Доказательства законов перестановочности, сочетательного и распределительного для действий `_+_` и `_*_` на области `Bin` (они просто выводятся из предыдущих двух пунктов).
- Структуры разрешимого линейно упорядоченного множества `DecTotalOrder` относительно упорядочения `≤_` («меньше или равно») на области `Bin`.

- Действия вычитания $_{-}$, с доказательством его определяющих свойств.
- Доказательства законов монотонности операций $_{+}$, $_{*}$, $_{-}$ относительно сравнений $_{<}$, $_{\leq}$.
- Действия `divMod` деления с остатком, с доказательством его определяющих свойств.
- Действия `gcd` вычисления наибольшего общего делителя, с доказательством его определяющих свойств.

Заметим, что первые два из перечисленных доказательств являются обоснованием замысла введения двоичного представления. Без них введение двоичного представления даже не имеет смысла. Что касается операций `divMod` и `gcd`, то они очень желательны для стандартной библиотеки, так как, например, используются в арифметике дробей, и кроме того, стандартная библиотека включает такие операции для области $\mathbb{N}$.

Вообще, какие доказательства должны быть в доказательной программе? Предположим, программа использует какую-то известную математическую операцию. Во многих случаях определяющие свойства этой операции входят в программу в виде объявления типа. Например, ниже в Разделе 4.5 представлено определение деления с остатком. В других случаях определением может быть некий алгоритм, не обязательно высокой производительности (как, например, сложение унарных записей). И доказательная программа по меньшей мере должна включать доказательство того, что применяемый алгоритм вычисления даёт итог, удовлетворяющий определению.

ПРИМЕР 6. *Если функция (программа) деления с остатком натурального числа a на натуральное число b даёт частное q и остаток r , то должно быть доказательство равенства $a = q*b + r$ и неравенства $r < b$.*

Сложение двоичных записей натуральных чисел должно давать тот же итог, что сложение унарных записей, — с учётом преобразования кодировок.

□

При программировании доказательной библиотеки компьютерной алгебры также желательно включать доказательства лемм и теорем известных из учебников. Такая библиотека должна быть не только собранием алгоритмов, но и формальным машинно-проверяемым

изложением теории, связанной с этими алгоритмами. Например, для программы сложения натуральных чисел желательны доказательства сочетательного и перестановочного законов.

3.3. Что даёт библиотека `Binary`

В библиотеке реализованы функции, нехватка которых отмечена в предыдущем разделе. Это даёт арифметику натуральных чисел, которая а) снабжена всеми необходимыми машинно-проверяемыми доказательствами, б) имеет операции такой вычислительной стоимости, какая известна для способа «в столбик». Например, сложение имеет порядок стоимости ограниченный линейной величиной относительно количества разрядов в аргументах, а умножение и деление с остатком имеют соответственно квадратичный порядок стоимости.

Приступим к изложению подробностей построения этой библиотеки.

4. Основные части разработки

4.1. Отношение порядка

Двоичные записи сравниваются в отношении `_<_` как списки разрядов (в отличие от сравнения в стандартной библиотеке). Только сначала сравниваются двоичные порядки операндов (число цифр): тот операнд больше, чей порядок больше. Если порядки равны, то списки разрядов сравниваются лексикографически, начиная со старшего разряда. Причём порядок вычисляется в унарной записи, его выдаёт функция `length` длины списка. Получается, что сравнение порядков имеет стоимость линейную относительно величины порядков операндов, поэтому оно не увеличивает порядка стоимости сравнения двоичных записей.

4.2. Изоморфизм отображения `toN` по сложению

Формальное доказательство на Агде взаимной однозначности двоичного кодирования основано на следующих построениях. Вводится функция

```
fromDigs : List Bit → N
```

перевода из списка двоичных разрядов в унарное представление. Для операций `suc` и `pred` доказываются леммы

Agda —

```

1 suc-assoc : (a b : Bin) → (suc a) + b ≡ suc (a + b)
2 pred◦suc : pred ◦ suc ≡ id
3 suc◦pred : (bs : List Bit) → suc (pred (bs 1#)) ≡ bs 1#

```

Лемма `suc-assoc` объяснена в разделе 2.6. `(bs 1#)` есть общий вид двоичной записи ненулевого числа. В стандартной библиотеке имя `id` обозначает тождественную функцию, а равенство функций $f \doteq g$ означает утверждение $\forall x (f x \equiv g x)$.

В этой части доказательств библиотека использует также второе каноническое представление числа — в виде списка разрядов (`List Bit`), как это объяснено в разделе 3.1.

Доказывается теорема об инъективности отображения `fromDigs`:

Agda —

```

1 injective-fromDigs : ∀ {bs bs'} → HasLast1if bs → HasLast1if bs'
2                      → fromDigs bs ≡ fromDigs bs' → bs ≡ bs'

```

То есть отображение `fromDigs` является инъективным на множестве тех списков разрядов, которые являются каноническими (либо пустыми, либо имеющими единицу на конце). Два условия с предикатом `HasLast1if` добавлены из-за того, что функция `fromDigs` применима к любому списку разрядов, а условие теоремы требует канонического представления числа.

Далее вводятся функции

Agda —

```

1 toBDigs : (n : ℕ) → ∃ (bs : List Bit)
2                      → (fromDigs bs ≡ n) × HasLast1if bs
3 toBDigs : ℕ → List Bit
4 toBDigs = proj₁ ◦ toBDigs

```

Первая из них преобразует число `n` из унарной записи в двоичную запись второго канонического вида для `n`. В результате получается пара `(bs, (eq , hl))`, где `bs` есть список разрядов, `eq` свидетельство равенства `fromDigs bs ≡ n`, а `hl` — свидетельство того, что `bs` есть каноническая запись.

Здесь и ниже `proj₁` и `proj₂` обозначают функцию проекции пары соответственно на первую и вторую составляющие.

Следующие две функции представляют доказательство того, что отображения `fromDigs` и `toBDigs` взаимно обратны на области канонических списков разрядов:

```

1 fromDigitsotoBDigs : fromDigs o toBDigs ≐ id
2 fromDigitsotoBDigs = proj1 o proj2 o toBDigs
3 toBDigsfromDigs : {bs : List Bit} → HasLastlif bs
4                                     → toBDigs (fromDigs bs) ≐ bs

```

Доказательство второй леммы использует `injective-fromDigs` и `fromDigitsotoBDigs`.

Сложение «столбиком» в двоичном представлении берётся из стандартной библиотеки. Оно основано на функции `addBitLists`, складывающей два списка разрядов и данный ряд *переноса*. Эта функция в свою очередь вызывает функцию `addCarry` прибавления переноса к числу, представленному списком разрядов. Функция `addC1` есть прибавление единичного переноса.

Доказательство гомоморфности по сложению отображения `toN` выражено следующей цепочкой лемм:

```

1 fromDigs-suc-eq : {bs : ListBit} → fromDigs (addC1 bs) ≐ 1+ (fromDigs bs)
2 toBDigs-suc-eq  : {n : N} → toBDigs (1+ n) ≐ addC1 (toBDigs n)
3 fromN           : N → Bin
4 fromN n = fromBits (toBDigs n)
5 toBits≐toBDigsotoN : a : Bin → a ≠ 0# → toBits a ≐ toBDigs (toN a)
6 fromNotoN         : fromN o toN ≐ id
7 toNofromN         : toN o fromN ≐ id
8 toN-suc-homo      : toN o suc ≐ 1+_ o toN
9 fromN-suc-homo    : fromN o 1+_ ≐ suc o fromN
10 toN+homo         : (a b : Bin) → toN (a + b) ≐ (toN a) +n (toN b)

```

Лемма `fromDigs-suc-eq` утверждает, что прибавление единичного переноса в двоичной записи соответствует прибавлению единицы в унарной записи при отображении `fromDigs`. Лемма `toBDigs-suc-eq` утверждает обратное соответствие.

Леммы `fromNotoN` и `toNofromN` утверждают, что отображения `toN` и `fromN` взаимно обратны между областями `Bin` и `N`. Их доказательства используют предыдущие построения.

Леммы `toN-suc-homo` и `fromN-suc-homo` утверждают, что отображения `toN` и `fromN` сохраняют операцию сложения с единицей.

Наконец, теорема `toN+homo` означает, что отображение `toN` сохраняет сложение. Здесь `_+n_` обозначает сложение в унарной системе (функция `_+_` из модуля `Nat` импортирована с переименованием в `_+n_`).

Из всего перечисленного выше легко выводится, что

- `toN` и `fromN` суть взаимно обратные изоморфизмы по сложению между областями `Bin` и `N`,

- операция сложения на области `Bin` обладает свойством сочетательности и перестановочности — это выводится из изоморфизма по сложению и из доказанных в стандартной библиотеке свойств сочетательности и перестановочности сложения в унарной арифметике.

Первая часть этих утверждений выражается на Агде такими тремя объявлениями типов:

```
1 ∀ (x y : Bin) → toN (x + y) ≡ (toN x) +n (toN y)    -- гомоморфность
2 toN ∘ fromN ≡ id
3 fromN ∘ toN ≡ id
```

Из второй части утверждений приведём выражение на Агде сочетательного закона:

$$\forall (x \ y \ z : \text{Bin}) \rightarrow (x + y) + z \equiv x + (y + z).$$

4.3. Изоморфизм отображения `toN` по умножению

Умножение «столбиком» в двоичном представлении берётся из стандартной библиотеки. Алгоритм состоит в повторении действий сдвига списка разрядов (то есть приписывания нуля спереди) и сложения двоичных записей `addBitLists`.

Свойство `toN*hom` гомоморфизма по умножению отображения `toN` доказывается в виде следующей цепочки лемм:

```
1 2bin*comm      : ∀ x → 2' * x ≡ x * 2'    -- где 2' есть двоичная запись для 2
2 *2-assoc      : (x y : Bin) → (x *2) * y ≡ (x * y) *2
3 2bin*assoc     : (x y : Bin) → (2' * x) * y ≡ 2' * (x * y)
4 2bin*assoc2    : (x y : Bin) → (x * 2') * y ≡ x * (2' * y)
5 *suc-even     : (x y : Bin) → (suc (x *2)) * y ≡ y + (x *2) * y
6 *suc          : (x y : Bin) → (suc x) * y ≡ y + (x * y)
7 toN*hom       : (a b : Bin) → toN (a * b) ≡ toN a *n toN b
```

Прямое доказательство свойства гомоморфности `toN*hom` отображения `toN` по умножению построить не легко. Поэтому сначала доказываются некоторые свойства умножения на 2, некие подобиya перестановочного и сочетательного законов (первые четыре из перечисленных лемм). Их последовательное доказательство просто, так как умножение на 2 имеет простое представление — приписывание нуля к списку разрядов.

Свойство `*suc` распределительности при умножении на $(1 + x)$ доказывается через представление числа в виде $(x * 2)$ или

$$1 + (x * 2)$$

(случаи «чёт» и «нечёт») и применение предыдущих лемм.

Наконец, свойство `toN*hom` гомоморфизма по умножению доказывается применением предыдущих лемм, теоремы `toN+hom` из раздела 4.2 и теоремы `*n-rDistrib` из стандартной библиотеки о распределительности для унарной арифметики. Сочетательность и перестановочность умножения на области `Bin` и распределительный закон легко доказываются перенесением этих свойств из унарной арифметики, для которой они доказаны в стандартной библиотеке. Это перенесение использует гомоморфизм `toN*hom`, и взаимную обратность отображений `toN` и `fromN`, упомянутую в разделе 4.2.

Добавив просто доказываемые законы нуля и единицы, получим для области `Bin` доказательную модель коммутативного полукольца `CommutativeSemiring`.

4.4. Вычитание

В библиотеке `Binary` используется следующее определение разности $x - (c + y)$ двоичных записей x и y и разряда заёма c . Это есть понятие усечённой разности (как в случае вычитания для унарных записей в стандартной библиотеке), учитывающее то, что разность должна быть натуральным числом.

```

1 module _ (c : Bit) (x y : Bin)           -- Подчерк после слова module означает
2   where                                   -- пропуск имени модуля.
3   c+y = (fromBits [ c ]) + y             -- то есть carry + y
4   data MinusRes (d : Bin) : Set
5   where
6   minuend< : x < c+y → d ≡ 0# → MinusRes d
7   minuend≡ : x ≡ c+y → d ≡ 0# → MinusRes d
8   minuend> : x > c+y → (x ≡ c+y + d) → MinusRes d

```

Это соответствует исходному определению разности $a - b$ как решению уравнения $b + x = a$. Итог вычитания имеет тип `MinusRes`. Параметр d этого типа означает разность $x - (y + c)$. Строчка `minuend>` описывает итог, когда $x > c + y$. Как видно из этой строчки, в этом случае требуется выполнение равенства $x \equiv c+y + d$. Предыдущие две строчки описывают итог в случае $x \leq c + y$, здесь требуется, чтобы разность d была равна нулю.

Вычитание в общем виде выполняется функцией

```

1 minus : (c : Bit) → (x y : Bin) → ∃ (\d → MinusRes c x y d)

```

Она возвращает пару (d, res) , где $(d : \text{Bin})$ есть собственно разность, а res содержит итог сравнения чисел x и $(c + y)$, и в случае $x > c + y$ ещё содержит свидетельство равенства $x \equiv c + y + d$. То есть кроме классического итога выдаётся свидетельство выполнения условия, являющегося определением вычитания. Все части этого итога программа вычисляет рекурсивно по построению списков разрядов в x и y . Доказательство использует несколько лемм, описанных выше, и теоремы монотонности умножения относительно неравенств $_<_$ и $_<=_$.

ПРИМЕР 7. Для двоичных записей $1' = \text{fromN } 1$, $2' = \text{fromN } 2$, $3' = \text{fromN } 3$ имеем

$$\text{minus } 0b \ 3' \ 2' = (1', \text{minuend} > 3' > 0 + 2' \ 3' \equiv 0 + 2' + 1').$$

Здесь выполнено

$$c = 0b, x = 3', y = 2', d = 1',$$

значение $3' > 0 + 2'$ есть свидетельство для неравенства $3' > 0\# + 2'$, $3' \equiv 0 + 2' + 1'$ есть свидетельство для равенства $3' \equiv 0\# + 2' + 1'$, а $0\#$ есть итог перевода нулевого разряда в нулевую двоичную запись. $\square$

Вычитание в узком смысле, — когда итог есть только двоичная запись, — задаётся функцией

```

1 _-'_ : Op2 Bin
2 x -' y = proj1 (minus 0b x y)  -- вычесть с нулевым заёмом и взять проекцию

```

Доказана лемма о том, что при отображении toN вычитание $_-'__$ соответствует вычитанию в унарной арифметике:

$$\text{toN-}'\text{homo} : \forall x y \rightarrow \text{toN } (x -' y) \equiv (\text{toN } x) -n (\text{toN } y).$$

4.5. Деление с остатком

Итог деления с остатком выражен в виде записи (`record`)

```

1 record DivMod (dividend divisor : Bin) : Set where
2   constructor result
3   field
4     quotient      : Bin
5     remainder     : Bin
6     equality      : dividend ≡ remainder + quotient * divisor
7     rem<divisor   : remainder < divisor

```

Это есть алгебраическое определение деления с остатком натурального числа `dividend` на число `divisor`. В нём `quotient` есть частное, `remainder` — остаток, `equality` — свидетельство для равенства «делимое = остаток + частное * делитель», `rem<divisor` — свидетельство для неравенства «остаток < делитель».

Функция `divMod` выдаёт итог деления с остатком, если делитель не равен нулю:

```
1 divMod : (dd dr : Bin) → dr ≠ 0# → DivMod dd dr
2 divMod dd dr = ...
```

Реализован алгоритм деления «столбиком». Он состоит в повторении действий сдвига (`shift`) списка разрядов делителя и вычитания `_'_'` двоичных записей:

повторять

`(n = max {i : ℕ | shift i dr ≤ dd}; s = shift n dr; dd := dd -' s)`

пока `dr ≤ dd`.

В этом цикле находится наибольшее `i` такое, что `s = 2i · dr ≤ dd`, и текущий остаток `dd` уменьшается на `s`. Видно, что количество действий над цифрами ограничено оценкой $O(|dd|^2)$, где `|dd|` есть длина двоичной записи делимого.

Алгоритм для функции `divMod` записан так, что свидетельства `equality` и `rem<divisor` для соответствующих свойств итога пересчитываются и накапливаются в ходе рекурсивных вызовов. Всё это составляет доказательную программу деления с остатком.

5. Строение библиотеки `Binary 3.2` в целом. Модули

Ниже *сущностью* называем тип или функцию из рассматриваемого модуля. Тип может также выражать математическое определение, а функция может выражать алгоритм или доказательство.

Основные использованные сущности из стандартной библиотеки таковы:

- Доказательства законов коммутативного полукольца для унарного представления натуральных чисел.
- Типы, отношения и операции для двоичных записей: `Bit`, `Bin`, `addCarry`, `addBitLists`, `+`, `*`, `≡`.

Основные сущности, разработанные в библиотеке `Binary 3.2`, таковы.

РИСУНОК 1. Зависимость частей библиотеки

- Определение упорядочения $_<_$ на области Bin через лексикографический порядок на списке бит и доказательство его свойств разрешимого полного упорядочения.
- P1. Доказательства взаимной однозначности и взаимной обратности отображений toN и fromN .
- P2. Доказательство изоморфизма отображений toN и fromN по сложению.
- P3. Доказательство изоморфизма отображений toN и fromN по умножению.
- P4. Вывод законов коммутативного полукольца для области Bin .
- Определение вычитания на области Bin и его реализация.
- Определение деления с остатком на области Bin и его реализация.
- P5. Доказательство определяющих свойств вычитания на области Bin .
- P6. Доказательство определяющих свойств деления с остатком на области Bin .

Здесь метками P1, ..., P6 обозначены соответствующие доказательства в библиотеке. Логическая зависимость между этими доказательствами приблизительно выражается диаграммой на рисунке 1.

Кратко опишем содержание каждого модуля. Порядок перечисления модулей согласован с импортом модулей в следующем смысле. Очеред-

ной модуль в этом списке ничего не импортирует из последующих модулей.

LtReasoning.agda — это небольшой модуль, принадлежащий библиотеке Ульфа Норе́лла:

U. Norell, библиотека функций на языке Agda для поддержки доказательства неравенств, исходный код, 2018, <https://gist.github.com/UlfNorell/102f8e59744de8a69f94b39f09f70c1d>.

Его функции применяются в библиотеке **Binary-3.2** для удобного использования законов рефлексивности и транзитивности при выводе неравенств.

Двоичное представление (тип **Bin**) импортировано из стандартной библиотеки.

NatProp0.agda содержит некоторые сущности для унарных натуральных чисел (тип **N**), которые нужны в библиотеке и которых нет в стандартной библиотеке.

List0.agda содержит небольшой набор сущностей для списков (тип **List A**) над произвольным типом, которые нужны в библиотеке и которых нет в стандартной библиотеке.

Bin0.agda содержит функции сложения и умножения двоичных записей, сдвига разрядов, возведения в степень, умножения на 2, функцию $\text{toN} : \text{Bin} \rightarrow \mathbb{N}$. Эти функции лишь немного изменены по сравнению со стандартными. Это нужно для удобства построения тех доказательств, которых нет в стандарте. Ещё дано определение отношения порядка $_<_$ для двоичных записей, существенно отличающееся от стандартного.

Bin1.agda содержит вывод различных свойств лексикографического упорядочения на списках бит. С помощью этих свойств в этом модуле выводятся структуры строгого полного разрешимого порядка $\text{<-strictTotalOrder}$ для отношения $_<_$ и полного разрешимого порядка <-decTotalOrder для отношения $_<=_$. Оба отношения для области **Bin**.

Вводится функция двоичного порядка ($|_| = \text{bitLength}$) — битовой длины двоичной записи. Доказываются леммы о связи двоичного порядка с некоторыми операциями.

Bin2.agda содержит доказательства того, что функции **toN** и **fromN** суть взаимно-обратные биекции между множествами **Bin** и $\mathbb{N}$,

что они сохраняют операцию сложения, что сложение обладает сочетательным (`+assoc`) и перестановочным (`+comm`) свойствами.

`Bin2a.agda` содержит построение структуры коммутативного моноида по сложению для области `Bin`:

```
commutativeMonoid : CommutativeMonoid _ _
```

`Bin3.agda` содержит доказательства различных свойств умножения, того, что отображения `toN` и `fromN` сохраняют умножение, доказательство сочетательного и перестановочного законов для умножения, распределительного закона для пары действий (`_+_`, `_*_`) на области `Bin`, некоторые свойства отображения $2^{\sim}$ степени двойки, некоторые свойства отношения `_|_` делимости на области `Bin`.

`Bin3a.agda` содержит построение структуры коммутативного полукольца (`CommutativeSemiring`) для области `Bin`.

`Bin4.agda` содержит доказательства законов монотонности действий `_+_`, `_*_`, `pred` относительно отношений порядка `_<_`, `_<=_` на области `Bin`, закона сокращения `+lCancel` для сложения, монотонности отображений `toN` и `fromN` относительно отношения порядка.

`Minus.agda` содержит функцию вычитания двоичных записей, доказательства различных её свойств, доказательства некоторых лемм об уменьшении двоичного порядка при вычитании.

`DivMod.agda` содержит определение понятия деления с остатком и функцию деления с остатком для двоичных записей.

`GCD.agda` содержит определение понятия наибольшего общего делителя (НОД) и функцию вычисления НОД для двоичных записей.

`DivModTest.agda` содержит тестовую программу для проверки быстродействия функции деления с остатком для двоичных записей.

`GCDTest.agda` содержит тестовую программу для проверки быстродействия функции НОД для двоичных записей.

6. О доказательстве завершаемости

Компилятор Агды следует правилам конструктивной математики. Приблизительно говоря, для него существуют только те объекты, для которых в программе задан алгоритм (функция) их вычисления. При этом компилятор должен «видеть» доказательство завершаемости алгоритма.

Во многих случаях компилятор сам находит доказательство завершаемости, исходя из обозрения рекурсивного вызова функции на структурно синтаксически меньшем аргументе. Например, функция `_+` из раздела 1 содержит предложение с рекурсивным вызовом:

$$(\text{suc } m) + n = \text{suc } (m + n).$$

В правой части функция `_+` применяется так, что её левый аргумент `m` синтаксически меньше левого аргумента `(suc m)` из левой части. Из этого компилятор выводит завершаемость данного алгоритма.

Но также часто встречаются завершающиеся алгоритмы, для которых доказательство завершаемости не выводится компилятором автоматически. В этих случаях программа должна быть подправлена так, чтобы компилятор нашёл такое доказательство. Библиотека `Binary-3.2` решает эту проблему следующим образом. Целевая функция вызывает вспомогательную функцию `aux`, имеющую добавочный аргумент: счётчик (`cnt : ℕ`) — натуральное число в унарном представлении. Функция `aux` определена рекурсивно, и в шаге рекурсии записано синтаксическое уменьшение счётчика $(\text{suc } cnt) \rightarrow cnt$. Таким образом компилятор выводит, что счётчик не может уменьшаться бесконечно. Кроме того, функция `aux` определяется так, чтобы при значении 0 для счётчика происходил выход из рекурсии. Этого достаточно для того, чтобы компилятор вывел доказательство завершаемости.

ПРИМЕР 8. Пусть в начале функция принимает список `xs` и счётчик `cnt = length xs` (длина списка), и пусть в рекурсивном вызове список `xs` заменяется на `(f xs)`. Пусть также выход из рекурсии в программе задан для случая `cnt = 0` или `xs = []`. Функция `aux` должна содержать доказательство того, что из условия `cnt = 0` следует условие `xs = []`.

Например, если в `aux` включено построение свидетельства неравенства `length (f xs) < length xs`, то из него можно доказать в программе, что при условии `cnt = 0` список станет пустым (ибо счётчик уменьшается не быстрее, чем укорачивается список). Этого достаточно для того, чтобы компилятор удостоверился в завершаемости. $\square$

Рассмотрим теперь функцию деления с остатком (раздел 4.5). При её наивном задании компилятор сообщает об отсутствии доказательства завершаемости. Действительно: где тот аргумент функции `divMod`, который синтаксически уменьшается при рекурсивном вызове после каждого соответствующего сдвига и вычитания?

Можно было бы применить перевод toN числа dd в унарную запись, и далее нетрудно доказать что каждое вычитание $\text{dd} - 's$ синтаксически уменьшает соответствующую унарную запись. Но перевод числа dd в унарную запись может привести к показательному росту стоимости вычисления цикла, входящего в программу divMod . Это противоречит одной из целей создания библиотеки.

На самом же деле в качестве счётчика годится величина $|\text{dd}|$ — порядок, то есть длина двоичной записи dd . Здесь $|\text{dd}|$ есть унарная запись. Чтобы использовать этот счётчик, в программу включено доказательство неравенства $|\text{dd} - 's| < |\text{dd}|$ для каждого шага рекурсии в функции divMod . Это даёт формальное доказательство завершаемости и притом не увеличивает оценки стоимости вычисления.

И Агда не сдаётся так легко: целевое неравенство оказывается не совсем очевидным. Это заметно по приводимому ниже примеру деления. Неравенство выводится в программе из следующих двух лемм (первая используется в доказательстве второй):

Agda —

```

1 |bbs1|≡|y|⇒|bbs1-y|<|bbs1| : (x y : Bin) → x > 1bin → | x | ≡ | y |
2                                     → | x -' y | < n | x |
3 y≤x,y*2>x⇒|x-y|<n|x| : (x y : Bin) → x > 1bin → y ≤ x → y *2 > x
4                                     → | x -' y | < n | x |

```

Первая лемма $|bbs1| \equiv |y| \Rightarrow |bbs1 - y| < |bbs1|$ утверждает:

«Если двоичные записи x и y имеют одинаковый порядок и $x > 1$, то порядок записи $(x -' y)$ меньше порядка x ».

Вторая лемма $y \leq x, y * 2 > x \Rightarrow |x - y| < n | x |$ утверждает:

«Если двоичная запись x больше единицы, $y \leq x$ и $2 * y > x$, то порядок записи $(x -' y)$ меньше порядка x ».

Здесь мы пропускаем подробности всех этих доказательств. Они содержатся в исходных текстах библиотеки `Binary`.

Связь этих лемм с алгоритмом divMod покажем на примере. Пусть в очередном цикле получены значения

Agda —

```

1 делимое      = 10001
2 делитель     = 110
3 dr'          = 1100 -- сдвинутый делитель
4 новое делимое = 101

```

Здесь двоичные записи перевёрнуты, чтобы записи имели вид привычный читателю: когда старшие разряды ставятся слева. Делитель не меняется к следующему циклу, а порядок делимого должен уменьшиться. К делителю применяется сдвиг на один разряд. Получилось число dr' меньше делимого. Но дальше сдвигать делитель нельзя, чтобы не получить число больше делимого. Затем делается вычитание $10001 - '1100$, и получается разность, имеющая порядок меньший порядка делимого. Лемма $y \leq x, y * 2 > x \Rightarrow |x - y| < n |x|$ даёт общее доказательство этого утверждения об уменьшении порядка делимого.

7. О других работах по данному предмету

Автор разрабатывал данную библиотеку арифметики двоичных записей натуральных чисел независимо от других работ, в качестве любопытного упражнения в доказательном программировании. Имелась также в виду возможность исправить соответствующую часть стандартной библиотеки. Оказалось, что имеются ещё две разработки по данному предмету на языке Agda: А. Алексеева (2013, <https://github.com/Rotsor/BinDivMod>) и М. Escardo (2016, <http://www.cs.bham.ac.uk/~mhe/agda-new/BinaryNaturals.html>). Также, как и ожидалось, доказательные программы двоичной арифметики давно имеются в библиотеке Coq [3].

Арсений Алексеев в 2013 году представил библиотеку Agda-программ, содержащую необходимые дополнения к стандартной библиотеке, названные в предыдущем подразделе. Но этот код не сопровождается никакими объяснениями или документацией. Поэтому в данной статье не только дается ссылка на исходные тексты библиотеки Binary, но и приводятся необходимые описания и пояснения к программам. Даются основные принципы реализации этой библиотеки. Реализация библиотеки Binary разработана независимо и существенно отличается от реализаций других библиотек, обсуждаемых здесь.

Martin Escardo (2016) применил другое представление двоичных записей. Двоичные записи строятся только применением конструкторов 0 , 1 , r , где 1 соответствует отображению $n \rightarrow 2n + 1$, а r соответствует отображению $n \rightarrow 2n + 2$ (здесь знак $\backslash$ обозначает лямбда абстракцию). Например, число 3 представляется записью $1 (1 0)$, что выражает разложение

$$3 = 2*1 + 1 = 2*(2*0 + 1) + 1.$$

Эта библиотека не совсем полна, отсутствуют действия деления с остатком и НОД. Опять: дан только код, нет статьи, нет объяснений.

Далее, среди систем доказательного программирования и их библиотек самой известной является система **Coq** [3] (proof assistant). Она содержит большую библиотеку формальных машинно-проверяемых доказательств для многих известных алгоритмов и теорем. Реализация двоичной арифметики натуральных чисел имеется в стандартной библиотеке системы **Coq**. Часть **BinNatDef** содержит алгоритмы арифметики двоичных записей, части **ZAriTh**, **NAriTh** содержат необходимые формальные доказательства для этой арифметики. В работе [9] описывается способ реализации средствами системы **Coq** задачи похожей на ту, что решается в нашей разработке. Только ни наша статья, ни [9] не относятся к теории вычислительной алгебры, а относятся к применению техники построения доказательств на данном языке программирования и при поддержке данной библиотеки. А эти инструменты в нашем случае сильно отличаются.

Отношение библиотеки **Coq.NAriTh** к нашей библиотеке **Binary** таково. Представление двоичной записи в стандартной библиотеке **Coq** отличается от представления двоичной записи в библиотеке **Binary**. В части **PAriTh** стандартной библиотеки **Coq** отдельно определяется тип **positive**, состоящий из двоичных записей положительных натуральных чисел, с конструктором **xH** (единица) и конструкторами

$$x0\ xI : \text{positive} \rightarrow \text{positive},$$

выражающими приписывание в младший разряд нулевого или единичного бита соответственно. Это ведёт к некоторым неудобствам в доказательствах свойств арифметики двоичных записей натуральных чисел, ибо вместо единого представления есть представление положительных чисел и конструкция добавления нуля. Отметим, что вместо этого недостатка в стандартной библиотеке **Agda** и в библиотеке **Binary-3.2** имеется тот недостаток, что приписывание нулей к старшим разрядам даёт списки бит, которые не соответствуют никаким числам. Это ведёт к усложнению формальных доказательств.

Главное же отличие состоит в парадигме языка программирования, в языке построения доказательств: язык **Agda** чисто функциональный, основан на «ленивом» вычислении по умолчанию, **Agda** не представляет отдельного языка для доказательств. Общей идеей этих двух систем является использование зависимых типов и опора на конструктивную математику [5].

Алгоритмы и доказательства в библиотеке `Binary` построены автором независимо.

Язык `Agda` имеет немалый круг пользователей. И независимо от того, как тот или иной метод реализован в других системах, представленная здесь разработка (наряду с программами А. Алексеева и М. Escardo) является, как надеется автор, некоторым вкладом в библиотеку Агды и в испытание системы `Agda`.

Автором также сделан следующий выпуск 4 библиотеки `Binary`. Он представляет улучшение реализации библиотеки за счёт упрощения алгоритмов и доказательств, и некоторой оптимизации алгоритмов деления с остатком и вычисления двоичного логарифма. Выпуск 4 опирается на другое представление двоичного числа. Оно задаётся тремя конструкторами данных:

`0#` — для нуля,

`2suc` — для произвольного числа вида $2(1 + n)$,

`suc2*` — для произвольного числа вида $1 + 2n$.

Как следствие, все алгоритмы арифметики и доказательства изменены, они стали проще, и часть `Bin` стандартной библиотеки никак не используется.

Данная статья больше, чем наполовину относится не к новому выпуску 4, а именно к выпуску 3.2 (другие алгоритмы, другие леммы в доказательствах, и так далее).

Подытожим отношение между пятью упомянутыми библиотеками: `Al` — А. Алексеева, `Esc` — М. Escardo, `CoqNat` — из `Coq`, `Binary-3.2`, `Binary-4`.

- `Binary-3.2` и `Binary-4` появились позже.
- `Binary-3.2` и `Binary-4` отличаются от остальных трёх независимой и другой реализацией алгоритмов и доказательств.
- В `Al` и `Esc` есть только код, а `Binary-3.2` и `Binary-4` снабжены описаниями и этой статьёй.
- `Esc` несколько неполна по функциям.
- Двоичное представление в `Binary-4` такое же, как в `Esc`, и сильно отличается от представления в других библиотеках.
- Двоичное представление в `Al` и `Binary-3.2` взято из стандартной библиотеки Агды.

Добавим следующее замечание. Испытывая программу `Binary 3.2`, А. Алексеев заметил следующий недостаток в алгоритме деления

с остатком (Раздел 4.5). При фиксированном делителе стоимость выполнения алгоритма квадратично зависит от битовой длины делимого. При другом способе вычисления (как в библиотеке А. Алексеева) эта стоимость линейная. Это важно с практической точки зрения.

В выпуске 4 библиотеки этого недостатка нет, так как само представление двоичной записи наталкивает на применение лучшего алгоритма в программе.

8. Заключение

В представленной здесь библиотеке программ `Binary 3.2` реализованы цели, названные во Введении 1.3: построены такие *доказательные* программы на языке `Agda` для арифметики двоичных записей натуральных чисел, которые имеют порядок стоимости вычисления такой же, как известные алгоритмы способом «в столбик».

Полученная программа исправляет и существенно дополняет часть `Bin` стандартной библиотеки `lib-0.16` языка `Agda`. В статье описаны методы, какими была сделана эта разработка.

Имеется недостаток: много сложных доказательств. Во многом это вызвано представлением числа в виде сумки со списком бит, причем списки, заканчивающиеся нулем, не задают никакого числа. Версия 4 библиотеки `Binary` (<https://github.com/mechvel/Binary-4>) устраняет этот недостаток. Формальные доказательства в ней упрощаются благодаря другому представлению натуральных чисел.

Дальнейший проект по данному предмету в части стандартной библиотеки едва ли имеет смысл для автора, ибо всё, что надо, сделано. Возможен лишь выбор между различными разработками в этой части, например, с учётом работ А. Алексеева (<https://github.com/Rotsor/BinDivMod>) и М. Escardo <http://www.cs.bham.ac.uk/~mhe/agda-new/BinaryNaturals.html>).

С точки зрения библиотеки арифметики для Агды, требуется разработка доказательных программ для арифметики натуральных чисел, представленных списками макроцифр, как об этом написано в Разделе 2.4. Это смежное поле деятельности относительно предмета данной статьи.

8.1. Благодарности

Автор благодарен Андрею П. Немытых и Антонине Н. Непейводе за полезные замечания по содержанию статьи.

Список литературы

- [1] U. Norell. “Dependently typed programming in Agda”, AFP 2008: Advanced Functional Programming, Lecture Notes in Computer Science, vol. **5832**, Springer, Berlin–Heidelberg, pp. 230–266. [↑](#)_{509, 520, 523, 528}
- [2] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003. [↑](#)₅₁₀
- [3] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, MIT Press, 2013. [↑](#)_{510, 518, 546, 547}
- [4] А. А. Марков. «О конструктивной математике», *Проблемы конструктивного направления в математике. 2. Конструктивный математический анализ*, Труды МИАН СССР, т. **67**, М.–Л., 1962, с. 8–14. [↑](#)₅₁₄
- [5] P. Martin-Löf. *Intuitionistic type theory*, Bibliopolis, 1984. [↑](#)_{514, 515, 547}
- [6] W. A. Howard. “The formulae-as-types notion of construction”, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.*, Academic Press, Boston, 1980, pp. 479–490. [↑](#)_{509, 514, 515}
- [7] H. B. Curry, R. Feys. *Combinatory Logic*. V. I, North-Holland, Amsterdam, 1958. [↑](#)₅₀₉
- [8] А. А. Карацуба. «Сложность вычислений», *Оптимальное управление и дифференциальные уравнения*, Сборник статей. К семидесятилетию со дня рождения академика Евгения Фроловича Мищенко, Тр. МИАН, т. **211**, Наука, Физматлит, М., 1995, с. 186–202. [↑](#)₅₁₅
- [9] N. Magaud. “Changing data representation within the Coq system”, TPHOLS 2003: Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. **2758**, eds. D. Basin, B. Wolff, Springer, Berlin–Heidelberg, 2003, pp. 87–102. [↑](#)₅₄₇
- [10] С. Д. Мешвелиани. «О зависимых типах и интуиционизме в программировании математики», *Программные системы: теория и приложения*, **5:3** (2014), с. 27–50. [↑](#)_{520, 523, 530}

Приложение I. Пример доказательства: сложение с нулём в двоичном представлении

Приведём код той части библиотеки Binary 3.2, которая необходима для выражения и доказательства утверждения для двоичной арифметики

$$\forall x \rightarrow 0\# + x \equiv x.$$

Тип `Bit` содержит два значения: двоичные разряды `0b` и `1b`. Тип списков таких разрядов обозначили (`List Bit`).

Agda —

```

1 data Bin : Set where
2     0#      : Bin
3     _1# : (bs : List Bit) → Bin
4
5 -- Set есть тип всех типов уровня 0.
6 -- В Агде функция может принимать и возвращать тип.
7
8 -- toBits и fromBits преобразуют двоичную запись между форматами Bin и
9 -- List Bit.
10
11 toBits : Bin → List Bit
12 toBits 0# = 0b
13 toBits (bs 1#) = bs ++ [ 1b ]
14
15 fromBits-aux : Bit → Bin → Bin
16 fromBits-aux b (bs' 1#) = (b :: bs') 1#
17 fromBits-aux 0b 0# = 0#
18 fromBits-aux 1b 0# = 1bin
19 fromBits-aux _ 1b = _
20
21 fromBits &: Bin → List Bit
22 fromBits [] = 0#
23 fromBits (b :: bs) = fromBits-aux b (fromBits bs)
24
25 Carry : Set
26 Carry = Bit
27
28 addBits : Carry → Bit → Bit → Carry × Bit
29 addBits 0b 0b b = (0b , b)
30 addBits 0b b 0b = (0b , b)
31 addBits 0b 1b 1b = (1b , 0b)
32 ...
33
34 addCarry : Carry → List Bit → List Bit
35 addCarry 0b bs = bs
36 addCarry 1b [] = 1b :: []
37 addCarry 1b (0b :: bs) = 1b :: bs
38 addCarry 1b (1b :: bs) = 0b :: addCarry 1b bs
39 addCarry _ 1b = _
40 addCarry _ (1b :: _) = _

```

```

41
42 mutual    -- 'mutual' означает, что следующие две функции
43           -- вызывают друг друга рекурсивно
44
45 addBL : Carry → List Bit → List Bit → List Bit
46       -- сложить списки разрядов двоичных записей
47       -- с переносом, по правилу сложения двоичных записей
48 addBL c []      bs2      = addCarry c bs2
49 addBL c bs1    []      = addCarry c bs1
50 addBL c (b1 :: bs1) (b2 :: bs2) = addBL-aux bs1 bs2 (addBits c b1 b2)
51 addBL-aux : List Bit → List Bit → Bit × Bit → List Bit
52 addBL-aux bs1 bs2 (c' , b') = b' :: (addBL c' bs1 bs2)
53
54 _+_ : Bin → Bin → Bin
55 a + b = fromBits (addBL 0b (toBits a) (toBits b))
56 -- Сложение: перевести в списки разрядов и сложить эти списки и нулевой перенос.
57 -- Из стандартной библиотеки:  xs ::r x = xs ++ [ x ]
58 --                               значит приставить в конец один элемент.
59 fromBits-bs:1 : fromBits o (r 1b) ≐ _1#    -- Лемма:
60 fromBits-bs:1 []      = refl    -- ∀ bs (fromBits (bs ::r 1b) ≐ bs 1#)
61 fromBits-bs:1 (b :: bs) =
62   begin
63     fromBits ((b :: bs) ::r 1b)           ≐ ( refl )
64     fromBits (b :: (bs ::r 1b))          ≐ ( refl )
65     fromBits-aux b (fromBits (bs ::r 1b))
66                                           ≐ ( cong (fromBits-aux b) (fromBits-bs:1 bs) )
67     fromBits-aux b (bs 1#)               ≐ ( refl )
68     (b :: bs) 1#
69   ■
70 0+ : (0# ++) ≐ id    -- Теорема о сложении с нулём:  ∀ (x : Bin) (0# + x ≐ x).
71                       -- (0# ++) обозначает функцию  \x → 0# + x.
72                       -- 0+ есть функция, которая строит свидетельство равенства
73                       -- 0# + x ≐ x   по любому данному x : Bin.
74 0+ 0#                = refl    -- случай нулевого аргумента
75 0+ ([] 1#)           = refl    -- случай единичного аргумента
76 0+ ((b :: bs) 1#)    =        -- случай любого аргумента большего единицы
77   begin
78     0# + ((b :: bs) 1#)                                     ≐ ( refl )
79     fromBits (addBL 0b (0b :: []) (b :: (bs ::r 1b)))      ≐ ( refl )
80     fromBits (addBL-aux [] (bs ::r 1b) (addBits 0b 0b b))
81                                           ≐ ( cong (fromBits o addBL-aux [] bs') (addBits-00 b)
82                                           )
83     fromBits (addBL-aux [] (bs ::r 1b) (0b , b))           ≐ ( refl )
84     fromBits (b :: (addBL 0b [] (bs ::r 1b)))               ≐ ( refl )
85     fromBits (b :: (addCarry 0b (bs ::r 1b)))               ≐ ( refl )
86     fromBits (b :: (bs ::r 1b))                             ≐ ( refl )
87     fromBits-aux b (fromBits (bs ::r 1b))
88                                           ≐ ( cong (fromBits-aux b) (fromBits-bs:1 bs) )
89     fromBits-aux b (bs 1#)                                   ≐ ( refl )
90     (b :: bs) 1#
91   ■
92 where bs' = bs ::r 1b

```

В доказательстве конструктор `refl` есть простейшее доказательство равенства вида $X \equiv X$ по аксиоме рефлексии. Например, в строке

$$0 + 0\# (\square 1\#) = \text{refl}$$

конструктор `refl` выражает доказательство равенства

$$0\# + (\square 1\#) \equiv \square 1\#.$$

Компилятор *сводит* (частично вычисляет) (normalization) левую и правую часть этого равенства и получает равенство

$$\square 1\# \equiv \square 1\#,$$

которое есть подстановочный случай равенства $X \equiv X$ — рефлексивного закона. Конструктор `refl` ссылается на это сведение. Сведение выполняется интерпретатором в силу программного кода из той области, которую компилятор обзорекает. Сведение выполняется «лениво» (как в языке Haskell). Так, в данном примере процесс сведения левой части применяет все образцы, которые участвуют в реализации функции `_+_` для данных аргументов. В ходе сведения компилятор получает последовательно такие выражения:

```

1 0# + (□ 1#)
2 fromBits (addBL 0b (toBits 0#) (toBits (□ 1#)))
3 fromBits (addBL 0b □ (1b :: []))
4 fromBits (addBL 0b □ (1b :: []))
5 fromBits (addCarry 0b (1b :: []))
6 fromBits (1b :: [])
7 □ 1#

```

Конструкция `begin ... ■` выражает цепочку равенств последовательных значений, где каждое значение равно в отношении `_≡_` следующему значению. Выражения этих значений перемежаются конструкцией вида `≡⟨ ... ⟩`. Это значит что равенство следующему значению выведено применением композиции функций, обозначенных внутри конструкции `≡⟨ ... ⟩`.

Замечательно, что конструкция `begin ... ■` есть композиция применений функций `begin_`, `■`, `≡⟨_⟩` из стандартной библиотеки, где они запрограммированы на Агде.

Приложение II. Индекс имён типов и функций

addBits	функция сложения трёх разрядов: перенос $+bit_1 + bit_2 \rightarrow (c', s)$	551, 552
addBL	функция сложения списков разрядов и разряда переноса	552, 553, 554
addBL-aux	вспомогательная функция для addBL	552
addCarry	функция сложения двоичной записи с разрядом переноса	536, 540, 551, 552, 553, 554
addC1	$= (addCarry\ 1b)$	536, 554, 556
Bin	тип двоичных записей	526, 527, 528, 529, 531, 532, 536, 537, 538, 539, 540, 542, 543, 551, 552, 554, 556, 557
Bit	тип двоичных разрядов 0b, 1b	530, 531, 534, 535, 540, 551, 552, 554, 556
Carry	$= Bit$	551, 552
cong	функция построения свидетельства равенства по аксиоме конгруэнтности $l \equiv r \Rightarrow (f\ l \equiv f\ r)$	511, 512, 513, 529, 530, 552
divMod	функция деления с остатком двоичных записей	533, 540, 544, 545, 554
DivMod	тип итога функции divMod	539, 540
fromBits	: List Bit $\rightarrow$ Bin функция преобразования двоичной записи из списка разрядов . . .	531, 536, 538, 551, 552, 553, 554
fromBits-aux	вспомогательная функция для fromBits	551, 552
fromDigs	функция перевода из списка цифр в унарную запись	534, 535, 536, 554, 556
fromDigits-suc-eq	доказательство равенства $fromDigs\ (addC1\ bs) \equiv 1 + (fromDigs\ bs)$ для любого списка разрядов bs	536
fromDigits-toBDigs	доказательство равенства $fromDigs\ (toBDigs\ n) \equiv n$ для любой унарной записи n	536
from $\mathbb{N}$	: $\mathbb{N} \rightarrow Bin$ функция перевода унарной записи в двоичную	532

fromN-suc-homo	$:\text{fromN} \circ 1_+ \doteq \text{suc} \circ \text{fromN}$	536
fromNotoN	$:\text{fromN} \circ \text{toN} \doteq \text{id}$	536
List	конструктор типа списков над любым данным типом	531 , 534, 535, 536, 542, 551, 552, 554, 556
$\text{minuend}<$	конструктор из типа MinusRes для случая «уменьшаемое меньше $c+y$ »	538
$\text{minuend}\equiv$	конструктор из типа MinusRes для случая «уменьшаемое равно $c+y$ »	538
$\text{minuend}>$	конструктор из типа MinusRes для случая «уменьшаемое больше $c+y$ »	538 , 539
minus	вычитание двоичных записей в общем виде . . .	539, 555, 557
MinusRes	тип итога функции minus	538 , 539, 555
N	тип натуральных чисел в унарной записи	511, 512, 516, 520, 521, 522, 523, 524 , 526, 527, 528, 532, 534, 535, 536, 540, 542, 554, 556
pred	функция предыдущего числа для двоичной записи	532 , 534, 535, 543, 555
$\text{pred}\circ\text{suc}$	доказательство равенства $\text{pred}(\text{suc } x) \equiv x$ для любой двоичной записи x	535
refl	конструктор свидетельства для равенства вида $X \equiv X$	511, 512 , 528, 552, 553
shift	функция сдвига двоичной записи на несколько разрядов	540
suc	конструктор «следующий» натурального числа, функ- ция «следующий» для двоичной записи	511 , 512, 513, 521, 522, 523, 525, 529, 530, 532, 534, 555, 556, 557
suc-assoc	доказательство равенства $(\text{suc } x) + y \equiv \text{suc}(x + y)$ для двоичных записей	529 , 535
$\text{suc}\circ\text{pred}$	доказательство равенства $\text{suc}(\text{pred}(\text{bs } 1\#)) \equiv \text{bs } 1\#$ для двоичной записи вида $\text{bs } 1\#$	535
sym	доказательство равенства по аксиоме коммутативности равенства $1 \equiv r \Rightarrow (r \equiv 1)$	529 , 530

- $\text{toBDigs} = \text{proj}_1 \circ \text{toBDigits} \dots\dots\dots 535, 536, 554, 556$
- toBDigits перевод унарной записи в список двоичных разрядов,
кроме списка возвращает два свидетельства . 535, 536, 556
- $\text{toBDigs} \circ \text{fromDigits}$ доказательство равенства
 $\text{toBDigs} (\text{fromDigs } bs) \equiv bs$
для произвольного канонического списка двоичных
разрядов $\dots\dots\dots 536$
- toBDigs-suc-eq доказательство равенства
 $\text{toBDigs} (1+ n) \equiv \text{addC1} (\text{toBDigs } n)$
для любой унарной записи $n \dots\dots\dots 536$
- $\text{toBits} : \text{Bin} \rightarrow \text{List Bit}$ функция преобразования двоичной
записи в список разрядов $\dots\dots\dots 531, 536, 551, 552, 553, 556$
- $\text{toBits} \equiv \text{toBDigs} \circ \text{toN}$ доказательство равенства
 $\text{toBits } a \equiv \text{toBDigs} (\text{toN } a)$
для ненулевой двоичной записи $a \dots\dots\dots 536$
- $\text{toN} : \text{Bin} \rightarrow \mathbb{N}$ функция перевода в унарную запись
 $\dots\dots\dots 530, 532, 536, 537, 538, 539, 541, 542, 543, 545, 555, 556$
- $\text{toN-suc-homo} : \text{toN} \circ \text{suc} \doteq 1+ _ \circ \text{toN} \dots\dots\dots 536$
- toN+ homo доказательство того, что toN сохраняет сложение 536, 538
- toN- 'homo доказательство того, что toN сохраняет вычитание . 539
- toN* homo доказательство того, что toN сохраняет умножение 537, 538
- $\text{toN} \circ \text{fromN} : \text{toN} \circ \text{fromN} \doteq \text{id} \dots\dots\dots 536$
- 2bin*comm доказательство равенства $2' * x \equiv x * 2'$ для дво-
ичных записей
 $\dots\dots\dots 537$
- 2bin*assoc доказательство равенства
 $(2' * x) * y \equiv 2' * (x * y)$
для двоичных записей $\dots\dots\dots 537$
- 2bin*assoc2 доказательство равенства
 $(x * 2') * y \equiv x * (2' * y)$
для двоичных записей $\dots\dots\dots 537$
- $*\text{suc}$ доказательство равенства
 $(\text{suc } x) * y \equiv y + (x * y)$
для двоичных записей $\dots\dots\dots 537$

- *2-assoc** доказательство равенства

$$(x * 2) * y \equiv (x * y) * 2$$
 для двоичных записей **537**
- *suc-even** доказательство равенства

$$(\text{suc } (x * 2)) * y \equiv y + (x * 2) * y$$
 для двоичных записей **537**
- $\exists$ конструктор типа конструктивного существования, см.
 пояснение к функции `minus` **535**, 539
- $\circ$ знак функции `_o_` композиции функций
 520, **535**, 536, 537, 552, 555, 556
- $\times$ знак конструктора `_x_` прямого произведения типов
 510, 535, 551, 552
- $+$ знак сложения `_+_` унарных записей, двоичных записей
 510, **511**, 512, 513, 518, 522, 523,
 525, 528, 529, 530, 533, 535, 536, 537, 538, 539, 540, 551, 552, 553, 555
- $-'$ знак вычитания `_-'_` двоичных записей, см. функцию
`minus` **539**, 540, 545
- $\equiv$ знак синтаксического равенства `_=_` 511, 512, 513, **528**,
 529, 530, 535, 536, 537, 538, 539, 540, 545, 551, 552, 553, 554, 555, 556
- $\neq$ знак отрицания синтаксического равенства `_=_` . . **520**, 536
- $\overset{\circ}{=}$ знак отношения `_o_` поточечного равенства функций
 $\forall x (f\ x \equiv g\ x)$ 520, **535**, 536, 537, 552, 555, 556
- 0#** двоичная запись нуля
 529, 530, 531, 536, 538, 539, 540, 548, 551, 552, 553
- 1+** функция `suc` на унарных записях, см. тип `Bin` **536**, 554, 556
- _1#** конструктор двоичной записи ненулевого числа, см.
 тип `Bin` **531**, 551, 552
- 0+** доказательство теоремы о сложении с нулём слева:
 $\forall (x : \text{Bin}) (0\# + x \equiv x)$ **529**
- +0** доказательство теоремы о сложении с нулём справа:
 $\forall (x : \text{Bin}) (x + 0\# \equiv x)$ **529**
- []** конструктор пустого списка **531**, 532, 551, 552, 553, 558
- ::** знак конструктора `_::_` приставления элемента к списку
 спереди **531**, 551, 552, 553, 558

++	идентификатор функции <code>_+_</code> соединения списков	531, 551, 552, 558
:: ^r	идентификатор функции <code>_::^r_</code> присоединения элемента к списку сзади (<code>xs ::^r x = xs ++ (x :: [])</code>)	552, 558
$ bbs1 \equiv y \Rightarrow bbs1 - y < bbs1 $	первая лемма об уменьшении порядка разности двоичных записей	545
$y \leq x, y * 2 > x \Rightarrow x - y < n x $	вторая лемма об уменьшении порядка разности двоичных записей	545, 546

Поступила в редакцию 03.07.2018

Переработана 19.12.2018

Опубликована 30.12.2018

Рекомендовал к публикации

А. В. Климов

Пример ссылки на эту публикацию:

С. Д. Мешвелиани. «Построение доказательных программ арифметики натуральных чисел в двоичном представлении». *Программные системы: теория и приложения*, 2018, **9:4**(39), с. 509–560.

10.25209/2079-3316-2018-9-4-509-560

http://psta.psiras.ru/read/psta2018_4_509-560.pdf

Об авторе:

Сергей Давидович Мешвелиани

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

0000-0002-4224-6178

e-mail: mechvel@botik.ru

CSCSTI 50.05.13

UDC 510.252, 004.432.42

Sergei Meshveliani. *Constructing provable programs for arithmetic of natural numbers in binary representation.*

ABSTRACT. Support for *dependent types* in a functional programming language **Agda** makes it possible to include machine-checked proofs in the program. The problem of provable inclusion of algorithms for arithmetic operations on natural numbers in binary representation is considered.

A library of evidentiary programs for “written calculation” algorithms, acting on bit lists has been built. It contains machine-verifiable proofs of the required properties of the applied algorithms, corrects and substantially supplements the Bin part of the lib-0.16 standard library of the **Agda** language. (*In Russian*).

Key words and phrases: computer algebra, binary code arithmetic, provable programming, **Agda** language.

2010 *Mathematics Subject Classification:* 03B35; 68T15, 03D60

References

- [1] U. Norell. “Dependently typed programming in **Agda**”, AFP 2008: Advanced Functional Programming, Lecture Notes in Computer Science, vol. **5832**, Springer, Berlin–Heidelberg, 2008, pp. 230–266. [↑](#)_{509, 520, 523, 528}
- [2] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003. [↑](#)₅₁₀
- [3] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, MIT Press, 2013. [↑](#)_{510, 518, 546, 547}
- [4] A. A. Markov. “On constructive mathematics”, *Problemy konstruktivnogo napravleniya v matematike. 2. Konstruktivnyy matematicheskiy analiz*, Trudy MIAN SSSR, vol. **67**, M.–L., 1962, pp. 8–14 (in Russian). [↑](#)₅₁₄
- [5] P. Martin-Löf. *Intuitionistic type theory*, Bibliopolis, 1984. [↑](#)_{514, 515, 547}
- [6] W. A. Howard. “The formulae-as-types notion of construction”, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.*, Academic Press, Boston, 1980, pp. 479–490. [↑](#)_{509, 514, 515}
- [7] H. B. Curry, R. Feys. *Combinatory Logic*. V. I, North-Holland, Amsterdam, 1958. [↑](#)₅₀₉
- [8] A. A. Karatsuba. “The complexity of computations”, *Proc. Steklov Inst. Math.*, **211** (1995), pp. 169–183. [↑](#)₅₁₅

- [9] N. Magaud. “Changing data representation within the Coq system”, TPHOLs 2003: Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. **2758**, eds. D. Basin, B. Wolff, Springer, Berlin–Heidelberg, 2003, pp. 87–102. [↑]₅₄₇
- [10] S. D. Meshveliani. “On dependent types and intuitionism in programming mathematics”, *Program Systems: Theory and Applications*, **5**:3 (2014), pp. 27–50 (in Russian). [↑]_{520,523,530}

Sample citation of this publication:

Sergei Meshveliani. “Constructing provable programs for arithmetic of natural numbers in binary representation”. *Program Systems: Theory and Applications*, 2018, **9**:4(39), pp. 509–560. (*In Russian*).

10.25209/2079-3316-2018-9-4-509-560

http://psta.psiras.ru/read/psta2018_4_509-560.pdf