On dependent types and intuitionism in programming mathematics

Sergei D. Meshveliani *

Program Systems Institute of Russian Academy of sciences, Pereslavl-Zalessky, Russia. email: mechvel@botik.ru

Abstract. It is discussed a practical possibility of a provable programming of mathematics basing on intuitionism and the dependent types feature of a programming language. The principles of constructive mathematics and provable programming are illustrated with examples taken from algebra. The discourse follows the experience in designing in Agda a computer algebra library DoCon-A, which deals with generic algebraic structures and also provides the needed machine-checked proofs.

This paper is a revised translation of a certain paper published in Russian in 2014.

 $\textbf{Keywords:} \ \ \text{constructive mathematics, computer algebra, machine-checked proof, dependent types, } \\ Agda$

1 Introduction

This paper is a certain revised translation of the paper [12] published in Russian in 2014.

It describes the experience in searching for the most appropriate tool for programming mathematical computation. The investigation is based on the practice of programming symbolic computations in algebra.

The goal was to find an universal programming language with possibly rich mathematical expressiveness, to explain its advantages with respect to other languages, and to test this tool on implementing a considerable piece of a real-world computer algebra.

The history of this search and program design experience consists of the three main steps. They correspond to the three language classes:

- 1. generic programming languages,
- 2. purely functional languages,
- 3. languages with dependent types.

Below it is explained of why the feature (1) is important. For example, the Haskell language [8] belongs to the classes (1) and (2). In this language the author has implemented in 1990-es the DoCon library for commutative algebra [13] [14]. It is presumed here that the notion of pure functionality, and the main features of the Haskell language, are known to the reader.

Finally, we consider the Agda language [1] [18], which belongs to the classes (1), (2), (3). Developing a workable implementation for such a complex language is a great technical problem. Currently, Agda is, mainly, a workable tool (with a great field remaining for desirable optimizations). In this paper we try to show that Agda fits best the needs of programming computation in mathematics.

About the sources on mathematics: the mathematical notions used in this paper can be found in [20] (algebra) and in [7], [5] (computer algebra).

The following discourse concerns mainly the ways to program mathematical computations and proofs in the Agda language.

^{*} This work is supported by the FANO project of the Russian Academy of Sciences, the project registration No AAAA-A16-116021760039-0.

1.1 Generic programming

Programs for mathematical computation often operate in different domains in a common way. For example, algebra textbooks present a simple algorithm to compute the greatest common divisor (gcd) for a pair of elements in any Euclidean ring E. There is an infinite set of domains that can be substituted for E. Such are: integer number domain \mathbb{Z} , the domain $\mathbb{Q}[x]$ of univariate polynomials with rational coefficients, the domain $(\mathbb{Z}/(p))[x]$ of univariate polynomials with coefficients modulo a prime integer p, and infinitely many other domains.

This approach (generic programming) is implemented in programming languages: there have appeared languages with abstract/polymorphic types, with type classes. In such a language a program like gcd is written once for all Euclidean rings. For example, in the Haskell language each class of algebraic domains (we call it here a generic algebraic structure) (group, ring, field, and such) can be expressed as a type class, and each concrete domain of this class is called an instance of this class. A generic structure (roughly speaking) pretends to be an abstract theory, and its instance pretends to be a model of this theory.

For example, a generic structure of groups by addition can be (partially) expressed by the declaration

Note that this is only a signature of a generic structure, other group laws cannot be expressed in Haskell.

Further, if the types a and b have instances of Group, then it is possible to define a Group instance for the direct product (a, b) by the laws of direct product of groups:

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

This pretends to be the functor of direct group product.

In some other languages (say, ML) there are possible other constructs to support generic programming.

The first profound approach with generic programming in mathematics was implemented in 1977 – 1990 in the Spad language and in the Axiom library for scientific computation [3] written in Spad. Further, it has been designed the Aldor language, as a refinement and extension for Spad, it even has the dependent types feature.

The DoCon library [13] [14] is written in Haskell. The most fundamental difference points of this language from Aldor are pure functionality, the "lazy" computation model, absence of dependent types (the last feature is negative).

There was an additional reason for designing the DoCon library in Haskell: the commercial status of the Axiom system in 1990-es.

1.2 The problem of a domain depending on a value

Consider the example: the domain $D = \mathbb{Z}/(m)$ of integers modulo m depends on the parameter m. There are known various computation methods for D, in which m is changed at run-time over the set of values which is not even known at the stage of compilation. Also the correctness condition of a method may depend on m. For example, the Gauss method to solve a linear system over D is correct only for a prime m.

Another example: the domain of integer matrices $\mathtt{m} \times \mathtt{n}$ is a *semigroup* by the matrix multiplication only if $\mathtt{m} = \mathtt{n}$. Generally, the set of valid instances for a domain often depends on a value being computed at run-time.

The type systems of the languages like Haskell and ML do not provide constructs to represent such domains in a fully adequate way.

On the other hand, in a mathematical textbook, it is possible to describe a computation that operates, for example, with the domain $D = \mathbb{Z}/(m)$, with changing the value of m in a loop, and applying different methods depending on whether m is prime or not. So that we see a certain inadequacy of the type systems.

To handle with this inadequacy, the DoCon library applies an explicit coding of a domain as an Haskell data — in addition to the set of instances related to the type. This evil complication is the cost paid for mathematical expressiveness of the library.

The approach of interpreter of the domain coding in checking the correctness conditions for a domain leads to that 1) this check is programmed by the library implementor, or by the user (not by the compiler), which approach is not reliable 2) this leads to that the condition break for a method may become detected only after many hours of computation.

The approach with dependent types differs in that the type checker operates with the type expressions like $\mathbb{Z}/(m)$ above by doing certain symbolic computation on such expressions at compiletime, with treating m as a variable. For example, the primality condition for m can be is expressed as a certain type depending on m. So that this way of processing type expressions corresponds adequately to setting algorithms in the mathematical textbooks as described above.

1.3 Dependent types. The investigation goal

There have been designed programming languages with dependent types, in which the above adequacy problem is solved: Aldor [2], Gallina — the language of the Coq system [6], Agda [1] [18]. In these languages a type may depend of a value, and computation with types can be programmed as well as with ordinary values (at least this is so in Agda).

In 2013, the author started to design the version DoCon-A of the DoCon library that bases on the dependent types feature, with using the Agda language. Agda is chosen due to the following reasons.

- It is purely functional.
- It has the "lazy" computation model.
- It is close to Haskell (roughly speaking, it is an extension of Haskell), and the previous library
 is written in Haskell, as well as a certain prover, which waits for its application to composing
 proofs in Agda programs.
- Such is a personal taste and experience of the library developer.

Initially, the library author considered dependent types only due to the problem of a domain depending on a value. But then, it occurs that this also brings in a powerful tool of a logical language. Dependent types give the three main advantages:

- adequate representation of a domain depending on values,
- automatic verification,
- the possibility to formulate notions that are "understood" by the type checker, and to include in a program statements about this notions, and formal proofs for this statements, which proofs are checked automatically.

The goal of the project is to

- rewrite the DoCon library in the Agda language,
- investigate on this example the practical possibilities of the approach of constructive mathematics and provable programming based on dependent types.

Of course, the part of "to rewrite" meets various problems related to the approach of constructive mathematics (intuitionism) [10] [11].

In the sequel the approach of intuitionism and dependent types is explained on examples, and there are described some design principles for the library ([15] and the manual of [17] give much more detail).

Below the DoCon-A 2.00 library is often called DoCon-A, or 'library'.

2 Some Agda features. Simple program examples

We need to explain certain lexical and graphical features of the language, because without this most Agda programs are not possible to percept.

The Agda parser reads symbols in the UTF-8 coding. This makes it possible to set to the program such identifiers as, for example, \mathbb{N} , \approx , $\not\approx$, \equiv , $\not\leq$, \circ , \bullet , \mathbf{x}^{-1} , \neg .

The UTF symbols are entered to the source via the text editor under the editor mode agda-mode. In this mode the editor shows correctly these symbols on the screen.

Names (identifiers) in a program are separated by blank, for example operator or a variable is separated by a blank.

Example: consider the code fragment

```
\texttt{m}: \mathbb{N} \texttt{m} = \texttt{foo1} \texttt{n} = \texttt{foo2} 2*\texttt{n} \ge \texttt{m}: 2 * \texttt{n} \ge \texttt{m} \quad -- \text{ declaration of membership to a type} 2*\texttt{n} \ge \texttt{m} = \texttt{f m n} \quad -- \text{ applying function } \texttt{f} \texttt{p} = \texttt{g} \ 2*\texttt{n} \ge \texttt{m} \qquad -- \text{ applying g to a value in the type } (2 * \texttt{n} \ge \texttt{m})
```

Here $2*n\geq m$ is a variable. It denotes a value in the type $2*n\geq m$. And the symbols '*', ' \geq ' in the expression of this type denote respectively an operator and a data constructor — because they are separated with blanks.

Note also that the name $2*n\geq m$ of a variable is self-explaining, the reader can guess from it what is the type of this value, and what is its meaning.

So: the lexical rules of Agda bring in a) special mathematical symbols, b) more mnemonic and sense for identifiers.

2.1 Example of a program: forming a rational number

Let a function $\mathbf{f}: \mathbb{N} \to \mathbb{N}$ implement a map of natural numbers. And consider the function

```
g : \mathbb{N} \to \text{Rational}
g n = record{ num = 1; denom = f n; denom\neq0 = f<n>\neq0 } where f<n>\neq0 = <proof>
```

that uses f for forming a rational number 1/(f n).

The language construct ':' means "(this value) belongs to this type".

'T \rightarrow U' means the type of all functions from the type T to the type U.

The type for rational numbers can be declared as the following record:

In this record, the fields num and denom are respectively the parts of numerator and denominator, denom $\not\equiv 0$ is the field for a witness (proof) for that denom is not equal zero (otherwise the fraction is not defined).

The term (denom \equiv 0) is a certain *type family* T, depending of a parameter ("index") denom. Any data d: T of this type is a witness for the statement ''denom = 0''.

All this means that the above line g n = ... implements forming of a rational number 1/(f n).

In the function g, the part <proof> needs to be a function (algorithm) that forms any element ne of type $nT = \neg$ (f $n \equiv 0$) (that is a proof for negation of the equality of $(f n) \equiv 0$). This formal proof uses the definition of the relation \equiv (in Standard library), definition for the function

f (which we omit), and it expresses a proof for the statement $f n \not\equiv 0$. It can be built (in the user program), for example, by induction by n.

The type checker checks that the algorithm for the ne value returns a data in the type nT. This is done by a certain symbolic computation with type terms (normalization of a term by the program equations, unification of terms, and such).

This check is done before run-time, it does not need giving n a concrete value. And this all expresses forming a proof for a certain statement in which n is under the universal quantifier.

2.2 Constructive version for logical connectives

In constructive logic, implication is represented as a map (algorithm) from some type T to some type U — a map between two domains of witnesses. If the types T and U represent the statements P and Q respectively, then the type $T \to U$ represents the implication $P \Rightarrow Q$. Because any function f of type $T \to U$ maps any witness f : T for P to a witness T is T for T.

The constructive conjunction of P and Q is represented as a direct product $T \times U$ of the types. A member of this product is represented as a pair data (t, u).

The constructive disjunction of P and Q is represented as the disjoint union of the types: $T \uplus U$. a member of this union is written as the data $(inj_1 t)$ or $(inj_2 u)$, where inj_1 and inj_2 are the constructors of imbedding into the union from the types T and U respectively.

 \times and \uplus are not language constructs, they are type constructors defined in Standard library (implemented in Agda).

The following example gives a more definite notion of provable programming with using dependent types.

2.3 Example: defining a semigroup

The semigroup notion can be expressed as

```
record Semigroup (A : Setoid) : Set where open Setoid A using (_{\approx}) renaming (Carrier to C) field _{\bullet} : C \rightarrow C \rightarrow C cong_{\bullet} : (x y x' y' : C) \rightarrow x \approx x' \rightarrow y \approx y' \rightarrow (x _{\bullet} y) \approx (x' _{\bullet} y') assoc : (x y z : C) \rightarrow (x _{\bullet} y) _{\bullet} z \approx x _{\bullet} (y _{\bullet} z)
```

Here Setoid is any set Carrier with any equivalence relation $_{\approx}$ defined on it. This definition is taken from Standard library. The relation $_{\approx}$ is implemented by the programmer for each particular setoid instance.

A semigroup is a setoid on which carrier C it is defined a binary operation $_\bullet_$ satisfying the laws of cong \bullet , assoc.

The field $cong \bullet$ means the congruence law — of that the relation $_{\sim} =$ and the operation $_{\sim} =$ agree. This property is expressed as a type of a function. Each function of this type (this needs to be an algorithm accompanied by a termination proof) is a proof for this law for the corresponding semigroup. The expression of this type contains the indices x y x' y'. The function $cong \bullet$ maps each such quadruple and witnesses for $x \approx x'$, $y \approx y'$ into a proof for the equality $(x \bullet y) \approx (x' \bullet y')$.

The signature for congo is a constructive representation for the statement

```
for all x, y, x', y' from C (if x \approx x' and y \approx y', then x \cdot y \approx x' \cdot y').
```

The types constituting the signature for the $cong \bullet$ value depend on the values x, y, x', y'. This allows to represent the statement about the operation $_\bullet_$ and relation $_\bullet_$ in the form of a certain type family.

Also it is used here a constructive representation for implication via the type constructor $_\rightarrow_$ as it is described in previous section.

The assoc field has the type of a witness for associativity for _•_.

2.4 Example: a semigroup of natural numbers

It is given a program defining a notion of semigroup. And the semigroup instance for natural numbers is implemented by the following program:

```
nat+semigroup : Semigroup Nat.setoid
nat+semigroup =
                  record{ \_\bullet\_ = \_+\_; cong\bullet = cong+; assoc = assoc+ }
    where
    _+: \mathbb{N} \to \mathbb{N} \to \mathbb{N}
                                                -- addition in unary system
             + n = n
    (suc m) + n = suc (m + n)
    _=n_= Setoid._\approx_ Nat.setoid
                                             -- equality on \mathbb N
    assoc+ : (x y z : \mathbb{N}) \rightarrow (x + y) + z =n x + (y + z)
                    yz = refl
    assoc+ 0
    assoc+ (suc x) y z =
                        begin ((suc x) + y) + z
                                                       =n[ refl ]
                               suc ((x + y) + z)
                                                       =n[ PE.cong suc (assoc+ x y z) ]
                               suc (x + (y + z))
                                                       =n[ refl ]
                               (suc x) + (y + z)
                        cong = < skip >
```

Let us comment this.

Natural numbers (of type \mathbb{N}) are written in unary coding, via the data constructor **suc** ("successor").

Nat. setoid is the setoid instance for \mathbb{N} imported from Standard library. Its carrier is \mathbb{N} .

The semigroup operation $_\bullet_$ is implemented as the addition $_+_$ on \mathbb{N} . The two equations for $_+_$ implement the algorithm to evaluate this operation.

But the type checker needs to check termination of the algorithm. It this case, it is done by a certain built-in procedure. In the second equation the first argument ${\tt m}$ for the operation + in the right-hand side is syntactically smaller then the first argument (${\tt suc}\ {\tt m}$) for + in the left-hand side. Hence, the type checker concludes that this recursion terminates.

The function assoc+ is a proof for associativity for the operation +. It is done by induction by the construction of the first argument. In the first equation, the data constructor refl means that the equality (0 + y) + z = n + (y + z) is proved by reducing its parts to the normal form according to the definition of the function +. This is a computation on the type terms, with presence of variables (y, z). If the two normal forms do not coincide, the type checker reports of that the refl data is not a needed proof. Otherwise the final proof is by applying the reflexivity law x = n.

In the second equation for assoc+, the right-hand side is a proof for the equality (($suc\ x$) + y) + z =n ($suc\ x$) + (y + z). It is represented by the three successive equality transformations. In each line, the construct =n[...] presents a composition of the laws which prove the equality of the value in the current line to the value in the next line.

Thus, the construct =n[PE.cong suc (assoc+ x y z)] denotes that the associativity law is applied to the subterm (x + y) + z in the current term, and then it is applied the congruence law for the suc constructor with respect to the equality =n.

Digression about operation congruence. Note that in algebra we always deal with a *theory* with equality. For example, if $x \approx y$, then $f x \approx f y$, $x + a \approx y + a$, ...

In most textbooks on mathematics this congruence law is presumed. But an Agda program needs to point explicitly, where congruence holds, and to operate explicitly with its witnesses (as it is shown above in the construct of PE.cong suc (...)). Otherwise the type checker would not accept a proof. And this is natural, because in an arbitrary program the result is not necessarily agreed

with the instance for the equality \approx used in this program, taking also in account that \approx is most often implemented by the programmer.

Let us return to the last proof in the example. In the construct of

begin_ is a certain function applied prefixly, \square a function applied postfixly, $_=n[_]$ _ in an infix function of three arguments (it is a function from Standard library renamed to $_=n[_]$ _, we skip this import in the above code).

The construct of (begin ... = n[...] ...] is not a language construct. This is only applying the three functions programmed in Agda in Standard library.

This approach with introducing for proofs certain functions with infix denotation has the effect of implementing various languages for proofs.

Agda has not any special language for proofs. Proofs are written in the same language as all the rest.

A considerable experience of the DoCon-A library [17] in composing proofs (see the manual) does not show the necessity of any special language for proofs. Still there is needed further experience to decide on this.

3 The DoCon-A library as implementation of a part of constructive mathematics

Constructive mathematics [10] puts that each object must be built by some given algorithm accompanied by a proof for termination. For example, the DoCon library implements the algorithms for linear system solving, finding Gröbner basis ([5], Appendix I), factoring a polynomial over certain coefficient domains, and many others. For all these methods, as well as for many other useful methods, composing a formal termination proof does not present any real problem.

3.1 About searching for a constructive proof

Termination by syntactic decrement. In numerous easy cases, the type checker derives termination itself by observing a recursive call and finding an argument that becomes syntactically smaller, in a certain appropriate ordering (similar as with the above rules for +).

Termination by counter. In many other cases this built-in proof does not work. The library uses in such cases introducing to the function an additional argument — a counter for "steps" of type \mathbb{N} (with the suc constructor). And a termination proof is obtained via comparison of the counter value with an appropriate size function value for a certain argument, when the counter decreases by one occurrence of suc with each recursion step.

(Proofs of this kind are often given in textbooks and papers).

Also there is a feature of *sized types*, but we do not consider here this tool.

Termination by unfeasible bound. Sometimes it is known that the given algorithm terminates, but a proof for termination is rather complex, this may be a solution for some great problem. And it is needed to apply this algorithm without setting its termination proof to the program (which proof may take, say, a thousand of source pages).

There is an elegant way out. Add to the function a counter argument cnt for the number of steps, and a bound B for cnt, so that cnt is decreased from B towards 0. And put B to be some *unfeasible* number, for example, 2 ^ (10 ^ 100). Then a termination proof for the modified algorithm is simple, and the two algorithms produce the same result for all input for which the number of steps is feasible

(The trick with unfeasible bound is known to me from an e-mail message by Ulf Norell).

Termination for semidecision. Various semidecision algorithms are sometimes useful in practice. For example, searching a proof for an equation for complex enough calculus. A way out for this may be adding a bound for the number of steps. Both bounds may have sense, a feasible bound and unfeasible one.

Postulating termination. If the programmer is lazy to design some messy termination proof, one can delay this proof for future by postulating termination for a particular function. This is done by setting the {-# TERMINATING #-} pragma strictly before the function declaration.

DoCon-A uses the TERMINATING pragma only in certain two places where such usage is not important. These places are normal form functions in EqProver/* and in some Read instances. The excuse for this is as follows.

- A proof for this termination can be provided in future.
- The EqProver functions run only at the stage of type checking, they are parts of so-called *tactics*.
- The Read instances is a new feature, currently it has a draft implementation.

Non-constructive existence. Some computations and proofs in mathematics may include non-constructive operations. Consider the discourse "As R is a Noetherian ring with unity, there exists a maximal ideal I in R, and this ideal is generated by some finite generator set G. Then, put the result h = f(G)". In classical mathematics, the above maximal ideal I does exist. But choosing such an ideal cannot be done by an algorithm, in general case.

As an example of most easy constructivity problem among the difficult ones, consider the Higman's lemma [9]:

For any infinite sequence w(k) of words in a finite alphabet there exist numbers i and j such that w(i) is a subword in w(j)

(that is w(i) is obtained from w(j) by deleting several positions, may be, no positions).

It is known (C. Nash-Williams) its short and non-constructive proof. And it was expected that a constructive proof will be much more complex and long. Later there have been obtained machine-checked proofs with using the systems Coq, Isabelle, and finally Agda: [4] [19]. Thus, the proof in Agda (for the case of a two letters alphabet) takes only about 80 non-empty lines of the source program.

Finally: the last tool to use for handling any problems with constructivity is the 'postulate' construct.

DoCon-A does not use such.

3.2 Proof by contradiction

Very often it is possible in constructive mathematics (and in the Agda language). This is so, for example, in such a case when the corresponding relation P is decidable — there is given an algorithm to solve P.

The library contains many proofs by contradiction for decidable relations. An important source of this decidability is decidability of the equality relation \approx , which is required for classical generic structures.

As an example of undecidable relation, consider the equality relation in some groups defined by several generators and quotiented by several equalities (the word problem in a finitely generated group).

In principle, it is possible to postulate the law of *excluded third* and to apply it in Agda proofs, relying on the classical logic.

But for a system dealing with algorithms, it is much more appropriate to keep constructivity as far as possible.

4 Summary: about advantages of provable dependently-typed programming

The theoretical base for programming with dependent types is the intuitionistic type theory by M. Loef [11].

As soon as dependent types are used, algorithms (and program) are joined with proofs. And this makes it possible a provable programming, when the principal properties of the algorithm are automatically checked by the type checker.

More definitely, dependent types provide the following possibilities.

- 1. To express a property P of the algorithm as a type T (depending on values), where the data constructors for T can be defined by the programmer.
- 2. To express a proof for P as a function that builds any value in T.
- 3. To join in the source program the algorithm and a proof for its main properties (chosen by the programmer), and to do it so that this does not effect the run-time performance,
- 4. To rely on automatic check of the proofs.
- 5. To automatically check many theorem proofs in mathematics (for statements are expressed as dependent types).

Also the last two points are important because

- most good textbooks have errors (which are often caused by typos),
- an error in a program that drives a device may cause heavy effects.

Let us sketch certain important features of formal proofs in Agda.

- 1. The type checker would not accept an erroneous on incomplete proof.
- 2. A proof is a *data* of the Agda language, and sometimes it has sense for a program to analyze the structure of the proof.
- 3. The type checker first searches for a proof by default, by normalizing the type expressions by the definitions of functions in the scope. For example, the type
 - (suc 0) + (suc (suc 0)) \equiv (suc (suc 0)) + (suc 0) is normalized by the Standard library functions to suc (suc (suc 0)) \equiv suc (suc (suc 0)), and the latter has a proof given by the standard constructor refl.
 - Proof by normalization often occurs sufficient, and even more often occurs not sufficient.
- 4. The 'postulate' construct can be set for a proof that the programmer is lazy to compose. The means "trust me, so far". Even if all properties are postulated, this still produces a program in which domains are represented more adequately than in a language without dependent types.
- 5. There are some functions from Standard library which help composing proofs, Also most Standard library functions are provided with the corresponding proofs.
- 6. It is desirable to add to the library more provers (written in Agda) which help composing proofs.
- 7. A proof in Agda program is formal and complete. A proof of a statement is represented as an implementation for the function having the goal signature. This function is formed as a composition of functions which are proofs for some lemmata.

4.1 Example: a program for sorting a list

This function has an additional argument: a decidable ordering structure for the domain of the list elements. An usual approach in dependently-typed programming is the following.

- 1. The notions of the relation _<_ being a total order and _<?_ being the corresponding decision for _<_ are written in the form of a type declaration. All this is written as a record of type DecTotalOrder.
- 2. The notion of a list being ordered by the relation _<_ is written in the form of a type declaration.
- 3. It is written an algorithm sort for sorting. The sorting function is applied as (sort dto xs), where the record dto: DecTotalOrder contains the instances of <, <? and witnesses for their above properties.

4. The function sort returns the record of type SortResult xs which has the fields of ys, ord-ys, mSetEq. ys is the result list, ord-ys is a witness for that ys is ordered, mSetEq is a witness for that xs and ys have the same multiset of elements.

See the file List/Sorting.agda in DoCon-A where it is programmed the merge method for sorting.

Let us note that without dependent types the property of the relation _<_ being a total order cannot be expressed, neither it is checked by the compiler. And in the case of unlucky implementation for _<_, for example, this relation may occur not transitive, and the result list may occur not ordered.

5 The current state of the DoCon-A project

The current DoCon-A 2.00 release [17] implements only a small subset of the *methods* from the DoCon library, this is a certain introduction to the future provable version of DoCon. But this introduction includes an adequate domain representation and full machine-checked justification of all the used algorithms and constructs. And this makes it a large program which tests on practice the possibility to express a real-world computer algebra library in Agda.

DoCon-A 2.00 [17] implements the following hierarchy of classical algebraic structures:

```
DSet
                     (a set with decidable equality _{\sim}, with conditional enumeration)
         *<- Magma
                     (a set with abstract binary operation _●_)
    Semigroup
                     --> CommutativeSemigroup
1
    -
                     --> CommutativeMonoid
-1
    1
        Monoid
1
                          CCMonoid
                                                   (with the cancellation law)
1
1
         1
         Group
                         {\tt Factorization Monoid}
                                                  (with factoring to primes)
         CommutativeGroup
     -> Ringoid
         *----> Ring
         1
         RingWithOne
         1
         CommutativeRing
         IntegralRing
                                                  (\forall x y \rightarrow (x*y \approx 0 \rightarrow x\approx 0 \text{ or } y\approx 0))
                                EuclideanRing
                                                  (division with remainder, etc.)
                                GCDRing
                                                  (with an algorithm for gcd)
                                                  (with an algorithm to factor to primes)
         FactorizationRing
                                                  (the prime|split property added)
         Unique FactorizationRing
         Field
                                                   (each nonzero has an inverse)
```

Figure 1. The tower of classical algebraic structures supported in DoCon-A-2.00.

The following features are implemented.

- The *domain constructors* of Natural (\mathbb{N}), Integer (\mathbb{Z}), direct product for semigroups, Fraction, UnivariatePolynomial, EuclideanResidue.
- For N there are implemented the instances of CommutativeMonoid for addition and multiplication, and UniqueFactorizationMonoid for multiplication for N\0 [16].
- For Z there are implemented the instances of EuclideanRing and UniqueFactorizationRing.
- For Fraction (over any unique factorization ring) there is implemented the Field instance, with certified optimized methods for arithmetic.
- An univariate Polynomial over any CommutativeRing is represented by a certain pair list ordered decreasingly by exponents. The corresponding _+_ operation is implemented, and there are proved its properties of associativity and commutativity.
- EucResidue is the constructor of the residue ring R/(b) for any Euclidean ring R and any its nonzero element b being not invertible. This constructor builds the instance of CommutativeRing in the general case, and it builds the instance of Field when b is detected as prime.
- The demonstration program demoTest/EucResTest.agda runs the examples of arithmetics in R/(b) for the instance of $R = \mathbb{Z}$.
- The extended GCD method is programmed for an arbitrary EuclideanRing.
- A rich sub-library (AList) is implemented for operations with lists, association, lists, multisets.
 It includes the merge sorting function with all the needed proofs.

- The notions of FactorizationMonoid, FactorizationRing, FactorizationIsUnique are defined, and there are proved many lemmata about them [16].
- For the binary-coded natural numbers (Bin) it is added a proof for bijectiveness of the coding (toDigits) and for its homomorphism with respect to the successor operation.
- It is implemented the binary method for powering in a monoid, with proofs. It uses the binary coding for the exponent.
- Certain special equational provers InMonoid, InSemiringWithOne, InCommutativeSemiring are implemented and used.
- All the classical definitions properties are formulated as types for the above notions and constructs (as in a textbook on algebra, only given in full), all the proofs are provided for the methods.
- The performance of the programs demoTest/EucResTest (for residue ring), demoTest/SortingTest (for merge sorting), demoTest/FractionTest (for fraction arithmetic) is nearly as in the DoCon system (under Glasgow Haskell).

Let us consider some details.

5.1 Computational cost of a proof

Proofs do not effect the run-time performance of a program, for a reasonably designed program. But they take

- a volume of the program source code,
- memory space and processor time at the stage of type checking,
- time and effort in composing proofs, currently it is great.

For example, type-checking the DoCon-A 2.00 library needs the minimum of 11 G byte of heap and takes 70 minutes on a 3 GHz personal computer

(for Agda 2.5.3, ghc-7.10.2, Debian Linux).

I think that the type check cost currently presents the main problem for Agda (it looks like the Coq system also has such).

And there are possible and desirable certain optimizations in the type checker, which would, probably, reduce the above cost about dozen of times.

Proof size On the example of the DoCon-A 2.00 library, it occurs that the text size of proofs in the source code is approximately five times larger than the size of the the corresponding textbook containing all the necessary rigorous constructive "humanly" definitions and proofs.

With further development of the prover tools (in the library part), the proof volume will become smaller.

5.2 Examples of what is proved

Let us list some proof examples implemented in DoCon-A 2.00.

- "An inverse in a group is unique, and it holds $(xy)^{-1} = y^{-1}x^{-1}$ ".
- There are proved various properties of the residue ring Q = R/(b) for any Euclidean ring R, depending on the value of b.
- There are proved the base properties of the extended GCD algorithm in any Euclidean ring (u a + v b \approx gcd a b).
- It is proved that in any FactorizationRing factorization uniqueness is equivalent to the property Prime|split = \forall p a b \rightarrow IsPrime p \rightarrow p | (a \bullet b) \rightarrow p | a \uplus p | b.
- It is proved that in any EuclideanRing it holds the property Prime|split. The proof uses the above properties of the extended gcd method. This brings the unique factorization property to any Euclidean ring with factorization, for example, to Integer.
- The correctness of a certain optimized method for summing fractions over a domain R is derived from the condition of the unique factorization ring for R.
- The last three proofs automatically produce a correctness proof for optimized fraction addition over any Euclidean ring with factorization, in particular, over Integer.

5.3 Tools to compose proofs

Usually a programmer "translates" a rigorous constructive proof from a textbook or a paper into a machine certificate.

It is remarkable that for the proofs in the current library there are sufficient only the three constructs for composing a reasonably looking proof:

- 1. normalization,
- 2. composition of functions,
- 3. recursion.

Here (1) is a proof by computation — by normalizing to the same term,

- (2) represents a proof by using a lemma,
- (3) represents a proof by induction by construction of an argument data.

5.4 Proof meaning

Some mathematicians have the following prejudice:

"Programs in the verified programming tools (like Coq, Agda) do not provide a proof itself, instead they provide an algorithm to build a proof witness for each concrete data".

I claim: they also provide a proof in its ordinary meaning (this is so in Agda, and I expect, the same is with Coq).

Let us illustrate this with the example of proving the statement

for all
$$n \ (n \le n)$$
.

for natural numbers. The relation $_\le_$ is defined on $\mathbb N$ so that a witness for it can be built only with applying the two data constructors (axioms):

```
z \le n — "0 \le n for all n", and s \le s — "if m \le n, then suc m \le s suc n".
```

(Syntax: $z \le n$, $s \le s$ are function names, as they are written without blanks).

For example: $s \le s$ ($s \le s$ $z \le n$) is a proof for the statement $2 \le 5$.

Consider the inductive proof for the goal statement.

If n = 0, then $0 \le 0$ is proved by the law $z \le n$. For a nonzero, it is needed to prove $suc n \le suc n$. By the inductive supposition, there exists a proof p for $n \le n$. And the law $s \le s$ applied to p yields a proof for $suc n \le suc n$.

The corresponding proof in Agda is

```
theorem : \forall n \rightarrow n \leq n theorem 0 = z\leqn theorem (suc n) = s\leqs (theorem n)
```

For each $n : \mathbb{N}$ the function theorem returns a value in the type $n \le n$, that is the corresponding witness.

The second pattern applies the function theorem recursively. This all provides a proof in the two meanings. (1) At the run-time, (theorem n) yields a proof for $n \le n$ for each concrete n. (2) The very algorithm expression for theorem is a symbolic expression that presents a general proof for the statement "for all n (n \le n)".

The algorithm (program) theorem is a symbolic expression (term), its parts depending on a variable n. This term is verified by the type checker statically — before run-time. And this is the same as checking an ordinary inductive proof. Reasoning by induction corresponds to setting a recursive call to the algorithm for forming a witness.

We see that (2) provides a real generic proof for the statement, while (1) provides a witness for each concrete n. Similar it is with all proofs.

5.5 The problem of setting proofs

About proof translation Many proofs in the current library have been obtained from known rigorous constructive humanly proofs by "translation" to Agda. And the time and effort for this translation occur somewhat 50 times greater than I expected (this depends on the skill of a person, though). This contradicts to our expectation of that the above translation needs to be more or less a mechanical procedure. This presents a certain question for us.

Not only translation In the literature we often meet non-constructive proofs that can be replaced with constructive ones. This replacement often needs a nontrivial invention. And such an invention is currently done by an human much simpler than by any prover, a prover will not help, at the current state of art.

Also even "rigorous" constructive proofs in literature usually have considerable gaps; these gaps can be filled by the reader in a way more or less evident to the reader and to the author. For a machine-checked proof, someone needs to fill these gaps with machine-checked proofs, and most often this filling is not automatic.

Libraries of lemmata Currently the main tool that helps composing proofs is *accumulating the library of lemmata*. This follows an usual approach to developing a theory in mathematics.

For example, the library AList provides proofs for many general-purpose lemmata for association lists. In particular, they help to prove that the multiset sum satisfies the laws of associativity and commutativity, and this is used further in operations with the factorization data structures.

Special provers At the current state of art, an automatic prover can be really useful in special problems, where it is known an algorithm for solving a problem. This frees the programmer from setting manually a great number of proof steps. For example, many Agda proofs for equalities in the DoCon-A library can be automatically reduced to applying the Gröbner basis algorithm (the method from [5] Appendix I, modified for integer coefficients as shown in [14]). Though, it needs to be designed a certain translator, similar to the translators in EqProver/*, but a more complex one.

For a more generic case, there can be applied various versions of the *completion method* (a part of the *term rewriting* theory) for deriving an equality from other equalities — even though it is a semidecision procedure.

About general provers But special provers cover (on average) may be only 1/3 part of the proof design effort. I do not expect that applying the existing *generic* provers can make it any essentially easier. This is for the following reason. Speaking informally, searching for a proof in an Agda program consists of the following parts.

- 1. Inventing replacement for non-constructive parts.
- 2. Breaking the goal to several lemmata.
- 3. Applying a fixed finite set of standard proof attempts (induction by the chosen value, considering cases for the chosen value, and some others), thus developing a search tree of attempts.
- 4. Applying special provers to the appropriate parts.

And it occurs so that a prover often helps essentially only in the part (4).

The parts (1) and (2) are done much simpler by an human.

Consider the part (3). Choosing the current attempt to try is done much better by an human. This also concerns choosing the right expression to apply induction on, choosing the right expression to apply considering cases for, and so on — all this is done much easier by an human.

It remains the part (4). Here special provers help a lot.

So that the situation is: the automatic part (4) covers about 1/3 of the whole formal proof invention effort (assuming that other parts often include sub-parts done by (4)), and for all the rest, provers do not help any essentially.

Example Consider the function rev for reversing a list, where rev is implemented via repeated concatenation ++. The goal is to prove that (rev o rev) is the identic map. Represent this theorem by the function revrev.

First the programmer needs to search for an humanly proof. It is natural to try a proof by induction by the construction of the argument list xs. The step of induction has the goal of deriving the equality rev (rev (x :: xs)) \equiv x :: xs from the equality (1): rev xs \equiv xs. The goal equality is normalized to the goal rev ((rev xs) ++ [x]) \equiv x :: xs.

This is a difficult point for a prover (an automatic one, or an human). Automatic provers usually continue to develop the search tree by various attempts. There is a small finite set of derivation rule kinds, one of such rules is induction by some chosen value. This leads a prover to infinity — unless it "guesses" to apply at this point *searching a lemma*. A lemma needs to be some statement L such that L is proved by the prover in a reasonable number of steps, and then, the goal is derived successfully from L (this needs several recursive calls of the prover, with giving it a bound for the number of the search steps).

The human intuition hints that this lemma needs to be some equality for the term rev ((rev xs) ++ [x]) in the goal. The intuition also hints to choose the equality rev (ys ++ [x]) \equiv x :: (rev ys), and further, to substitute (rev xs) for the variable ys. Let us call this lemma rev-append. This lemma is easy to prove by induction by ys. And this lemma fills the gap in the goal proof, it only remains to apply the goal statement recursively: rev (rev xs) \equiv xs.

Translating the above found humanly proof to Agda yields the following program (this is also an example of setting proofs in an Agda program):

```
open import Relation. Binary. Propositional Equality as PE using (_{\equiv})
open import Data.List using (List; []; _::_; [_])
open PE.\equiv-Reasoning renaming (\_\equiv\langle\_\rangle to \_\equiv[\_]; begin_ to \equivbegin_; \_\square to \_\equivend)
module _ {a} (A : Set a)
  where
  _++_ : List A 
ightarrow List A 
ightarrow List A
                                                -- concatenation
  [] ++ ys = ys
  (x :: xs) ++ ys = x :: (xs ++ ys)
  \mathtt{rev} \; : \; \mathtt{List} \; \mathtt{A} \; \to \; \mathtt{List} \; \mathtt{A}
                                                -- reversing a list
  rev [] = []
  rev (x :: xs) = (rev xs) ++ [x]
  rev-append : \forall x ys \rightarrow rev (ys ++ [x]) \equiv x :: (rev ys)
                                                                                  -- lemma
  rev-append x []
                          = PE.refl
  rev-append x (y :: ys) =
       \equivbegin
         rev ((y :: ys) ++ [ x ])
                                                \equiv[ PE.refl ]
         rev (y :: (ys ++ [ x ]))
                                               \equiv[ PE.refl ]
         (rev (ys ++ [ x ])) ++ [ y ] \equiv [ PE.cong (_++ [ y ]) (rev-append x ys) ]
         (x :: (rev ys)) ++ [y] \equiv [PE.refl]
         x :: ((rev ys) ++ [ y ])
                                              \equiv[ PE.refl ]
         x :: (rev (y :: ys))
  \texttt{revrev} \;:\;\; \forall \;\; \texttt{xs} \;\to\; \texttt{rev} \;\; (\texttt{rev} \;\; \texttt{xs}) \;\equiv\; \texttt{xs}
                                                                      -- the goal theorem
  revrev [] = PE.refl
  revrev (x :: xs) = \equivbegin rev (rev (x :: xs))
                                                                   \equiv[ PE.refl ]
                                  rev ((rev xs) ++ [ x ])
                                                                    \equiv [ rev-append x (rev xs) ]
                                  x :: (rev (rev xs))
                                                                    \equiv[ PE.cong (x ::_) (revrev xs) ]
                                  x :: xs
```

≡end ------

Here the PE.refl proof step means (according to the general step of the proof by normalization) applying normalization of a term by the equations of the function rev.

How could an automatic prover help essentially in composing the above prove?

All the points are easy for the programmer, except searching for a lemma. But this part is done much easier by an human.

There are some provers based on many-sorted term rewriting which include the step of searching a lemma in the form of equality. There is a method for rejecting fast most of useless lemma candidates. Still the cost of the search—through is enormous. One of such provers finds an useful lemma after 10 minutes (on a 3 GHz machine). Still this part is easier for an human. Also this success depends on the initial choice of the operator set detected as related to the goal (this is a certain heuristic). This luck is not stable, and in most real examples an automatic prover will not help.

Summing it up: provers are useful in the part (4) of special provers — which is essential, but a large part of (1), (2), (3) is practically unaccessible for automatic provers — at the current state of art.

The problem of automating these parts (as possible) is principal one for the area of provable programming, and the most difficult one.

References

- 1. Agda. A proof assistant. A dependently typed functional programming language and its system. http://wiki.portal.chalmers.se/agda/pmwiki.php.
- 2. S. Watt, et al. *Aldor Compiler User Guide*. IBM Thomas J. Watson Research Center, http://www.aldor.org.
- 3. R. D. Jenks, R. S. Sutor, et al. Axiom, the Scientific Computation System. Springer-Verlag, New York–Heidelberg–Berlin, 1992.
- 4. Stefan Berghofer. A constructive proof of Higman's lemma in Isabelle. Types for Proofs and Programs, International Workshop (TYPES 2003), LNCS, volume 3085, pages 66–82, Springer-Verlag, 2004.
- Computer algebra. Symbolic and algebraic computation. Collection of papers edited by B. Buchberger,
 G. E. Collins, and R. Loos in cooperation with R Albrecht. Springer-Verlag, Wien 1982, New York 1983.
- 6. The Coq Proof Assistant. http://coq.inria.fr.
- 7. J. Davenport, Y. Siret, E. Tournier. Calcul formel. Masson, Paris, New York, 1987.
- 8. Haskell 2010: A Non-strict, Purely Functional Language. Report of 2010. http://www.haskell.org
- 9. Graham Higman. Ordering by divisibility in abstract algebras.

 Proceedings of the London Mathematical Society. 1952, 326–336, Volume s3-2, No 1.
- A. A. Markov. On constructive mathematics, In Problems of the constructive direction in mathematics.
 Part 2. Constructive mathematical analysis, Collection of articles, Trudy Mat. Inst. Steklov., 67, Acad. Sci. USSR, Moscow-Leningrad, 1962, pages 8–14.
- 11. Per Martin-Löef. Intuitionistic Type Theory. Bibliopolis. ISBN 88-7088-105-9, 1984.
- 12. S. D. Meshveliani. On dependent types and intuitionism in programming mathematics, (In Russian) In electronic journal Program systems: theory and applications, 2014, Vol. 5, No 3(21), pages 27–50, http://psta.psiras.ru/read/psta2014_3_27-50.pdf
- 13. S. D. Mechveliani. Computer algebra with Haskell: applying functional-categorial-'lazy' programming. International Workshop CAAP-2001, Dubna, Russia,
 - http://compalg.jinr.ru/Confs/CAAP_2001/Final/proceedings/proceed.pdf, 2001, pages 203-211.
- 14. S. D. Mechveliani. *DoCon. An algebraic domain constructor*. The source program and manual. Pereslavl-Zalessky, Russia. http://www.botik.ru/pub/local/Mechveliani/docon/
- 15. S. D. Meshveliani. *Programming basic computer algebra in a language with dependent types*, In electronic journal Program systems: theory and applications, 2015, 6:4(27), pp. 313-340. (In Russian), http://psta.psiras.ru/read/psta2015_4_313-340.pdf
- 16. S. D. Meshveliani. Programming computer algebra with basing on constructive mathematics. Domains with factorization. In RUSSIAN. In electronic journal Program systems: theory and applications, 2017, Vol 8, No 1, 2017, 44 pages, http://psta.psiras.ru/read/psta2017_1_3-46.pdf
- 17. S. D. Meshveliani. DoCon-A a provable algebraic domain constructor, the source program and manual, 2017, Pereslavl-Zalessky.
 - http://http://www.botik.ru/pub/local/Mechveliani/docon-A/

- 18. U. Norell, J. Chapman. Dependently Typed Programming in Agda, available at http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf
- 19. Sergei Romanenko. A proof in Agda of Higman's lemma. https://github.com/sergei-romanenko/agda-miscellanea/tree/master/Higman.
- 20. wan der Waerden. Algebra. Volume I. Springer, 1991. Algebra. Volume II. Springer, 1991.