Dependent Types for an Adequate Programming of Algebra

Sergei D. Meshveliani *

Program Systems Institute of Russian Academy of sciences,
Pereslavl-Zalessky, Russia. `http://botik.ru/PSI`

Abstract. This research compares the author's experience in programming algebra in Haskell and in Agda (currently the former experience is large, and the latter is small). There are discussed certain hopes and doubts related to the dependently typed and verified programming of symbolic computation. This concerns the 1) author's experience history, 2) algebraic class hierarchy design, 3) proof cost overhead in evaluation and in coding, 4) other subjects. Various examples are considered.
Keywords: dependent types, computer algebra, functional language, `Agda, Haskell`.

1 Introduction

The author has a considerable experience in computer algebra, in provers based on term rewriting, and in programming this in `Haskell` [9], [10], [6]. But he is a newbie to the *dependently typed programming* [11], [1], [4], [8]. This paper contains the considerations and questions about the possibility of a workable computer algebra library based on the *dependently typed and verified* programming in `Agda`.

In 1995 – 2000 the author has been developing a computer algebra library `DoCon` [9], [10]. It is written in the `Haskell` language [6] and uses the tool of `Glasgow Haskell` [5]. The aim is to program algebra in a generic style, with defining the classical categories of `Group, Ring, Field`, and so on, and their instances for the classical *domain constructors*: `Integer, Fraction, Polynomial, ResidueRing`, and the such. The goal was to implement this approach to programming algebraic methods by using a *purely functional language*, having a data class system, and with making this library open-source.

1.1 Dynamic Parameter Domain

The most problematic point in the `DoCon` project is the subtle feature of modelling a *domain depending on a parameter*, especially when this parameter needs to be evaluated at the running time.

* This work is supported by the Program No.16 of Fundamental Research of the Russian Academy of Sciences Presidium ("Fundamentalnye Problemy Systemnogo Programmirovaniya").

Example 1. The polynomial domain `P = Pol Rational vars` over rational coefficients has very different properties, depending on the `length n` of the variable list `vars`. For `n = 1`, `P` is an Euclidean ring, and needs to be provided with the instance of the EuclideanRing class — the one of division with remainder, with a certain classical properties satisfied. And for `n > 1`, the instance of EuclideanRing is not algebraically correct for `P`. And there are many computational methods, where the list `vars` is changed during evaluation.
Example 2. Consider the `Residue` domain `R/I` for `R : CommutativeRing`, `I` — an ideal in `R`. Most often `I` is defined by a finite list `gs` of generators. The simplest example is the residue domain `R' = Integer/(n)` – "integers modulo `n`". `R'` occurs a `Field`, if `n` is *prime*. There are many classical methods which are correct for `R'` being a `Field` (that is — for a prime `p`) and incorrect otherwise. Again, there are known methods where `n` changes during computation, and it is not known ab initio how many values will be sufficient. Such is, for example, the Chinese remainder method.

Hence, we cannot represent such a parametric *domain* as only a *set of* `Haskell` *class instances*. Because the `Haskell` types and instances are static.

As a way out, `DoCon` applies the *sample argument approach* [9], [10], which uses a certain symbolic coding of a domain into an `Haskell` data, with inserting these codes into each domain element representation. This (necessary) approach complicates the design essentially. In particular, the dynamic part of the domain check is not by the type check of `Haskell`, it is by the `DoCon` library code, and what it remains is on the user program.

Standard `Haskell` Algebra Classes.

In the late 1990-ies, the Haskell e-mail list had a huge discussion about reorganizing the standard library algebra classes. I wrote that there is not possible any more sensible reorganization than following the line of the domain coding (like it is in `DoCon`, and in its simplified standard library project called "Basic Algebra Library"). The reason for this is the above *dynamic parameter domain* problem. As a result of the discussions, the standard `Haskell` algebra hierarchy remains the same for today. Lennart Augustsson has noticed that the problem of a dynamic parameter domain can be solved in a language with *dependent types* [2].

In 1999 – 2001 I failed to find a workable system with dependent types May be, `Coq` [4] was such, but a) I have somehow missed it, b) its language is not close to `Haskell`. After the 11 year pause, I observed the situation by new — and discovered at least two working tools: `Coq` and `Agda`. Currently I am investigating the `Agda` possibilities [1], [11], [8], because it is easier to reformulate `DoCon` in `Agda` (in particular, I prefer 'laziness' on default). How will it look the `DoCon` library when formulated in `Agda`, with modelling domains exactly by dependent types, classes — by dependent records, and with adding proofs? The aim can be formulated as: an

adequate functional programming system and library for algebra (mathematics).

The two next features arise automatically from the approach:

1.2 Constructive Mathematics, Proofs

Let us note that rigorously defining types in an `Haskell` program cannot guarantee the program correctness. Consider, for example, programming the list sorting function, applied as `(sort (<) xs)`, and having an user-defined element comparison function `(<)` as argument. One cannot give `[2, (+)]` for `xs`, the compiler will check this out. But if one implements `(<)` so that it does not satisfy the *transitivity* property, the result list may occur not ordered. And this property of `(<)` is not checked by the compiler.

My first attempt to join a prover to an algebra library was by applying the techniques of *equational theories, many-sorted term rewriting*, a certain unfailing completion procedure [7], with adding a support for proofs in the predicate calculus. All this has been programmed in `Haskell` as a certain *prover* library. The main drawback of this approach is practical — of its *object language*.

1. The proofs are only for the programs written in a many-sorted term rewriting language — which is much more poor than `Haskell`.
2. The termination proofs are under a great question.

Now, with dependent types [1], [11], a programmer expresses adequately the above restrictions on arguments, they are checked by compiler (its type checker part).

Further, proofs appear in a program due to that (1) types may depend on values, (2) the truth of a statement is expressed by constructing any element of the corresponding type. The point (2) leads to the approach of constructive mathematics.

The problem of an object language is removed: proofs are for the programs in the same (very rich) language.

In the below discourse I assume that the reader is familiar with the concept of constructive evaluation and programs carrying proofs [11].

2 Trying Counter-Examples

Let us try to break in practice the "proved computation concept" of dependent types (let us call it briefly "DT (practical) concept").

Abbreviation: DT — dependent types.

We need a simple example which reveals an unnatural evaluation cost or type-checking cost for a program which mixes proofs with the "usual" evaluation. If we do not find such an example, we would be encouraged in advancing with the DT library for mathematics. Here follow several my naive attempts and considerations.

Why do we need to search for unusual effects — what is particular in the DT constructive approach? These are as follows.

(a) Proofs are data. And proofs are often computed as parallel to the 'ordinary' (non-proof) data computation in a loop.

(b) In 'human' mathematics, we need only a general proof for an algorithm, the one obtained *before* running computation. In the constructive DT model, applying an algorithm usually needs a witness for a proof for some property of a concrete argument. And this witness is built for each concrete argument value. And it is sometimes built at the running time — despite that the type check is done (in `Agda`) only before the running time.
(c) Proofs (witnesses) are often built as parallel to computing the ordinary data parts in a loop, so that ordinary computation data and the witness parts depend on each other. Programs are often formulated this way.

Example for the point (b): consider sorting a list `xs` of natural numbers, and suppose that orderedness of `ys` is a condition for applying `(f ys)` (that is otherwise the result of `f` may be incorrect or senseless). *In a classical computation*, its usage is like this: `ys = sort xs;` `zs = f ys`.
A proof for the statement $\forall$ `xs (IsOrdered (sort xs))` is generic, and is given somewhere separately of the program. The compiler does not check this proof.

In the DT constructive model, its usage is often like this:

```
r = sort xs;   zs = f (list r) (ordProof r)
```

Here `f` has an additional argument — a witness of that the first argument is ordered. The function `sort` returns the record `r`, which field `'list'` is the resulting list, and `ordProof` is a witness for orderedness of the `'list'` part. A program for constructing this witness is a part of the source program for `sort`, it is verified by the type checker before the running time. Still the value for this witness is sometimes built for a concrete list at the running time.
Question aside
Why is there used a concrete witness data while the general proof is already checked? Probably, this is due to the following reasons.

- This does not restrict the tool for the goal "compute and verify".
- If we skip the second argument in the above function `f`, then the language becomes so that it is difficult (or impossible) for the compiler to check the correctness of applying this function.
- A witness data for one part can be analyzed, and the program can use a part of this witness to form fast a witness for some other correctness condition. See, for example, the functions for proofs with the relation $m \leq n$ in the `Data/Nat` module and directory in Standard library for `Agda`.

The features (a), (b), (c) cause various practical questions. For example, *Does the verified evaluation necessarily increase the cost order of ordinary evaluation in some examples?* For example, one computes some problem in $O(n^2)$ steps, then applies the program version that carries verification in it, and the latter is evaluated in $O(n^4)$ steps. Is this possible?
(the effect also depends on how the proof part is used).

2.1 Objection Attempt 1

Example of sorting program for a list of natural numbers
Define the type `Ordered xs` expressing the statement of that a list `xs` is ordered non-decreasingly:

```
data Ordered? : List ℕ → Set
  where
  nil    : Ordered []
  single : (x : ℕ) → Ordered (x :: [])
  prep2  : (x y : ℕ) → (xs : List ℕ) → x ≤ y → Ordered (y :: xs) →
                                              Ordered (x :: y :: xs)
```

(see, for example, [11] for introduction to programming in `Agda`). Here the `nil` data constructor defines that the empty list is ordered, the `prep2` constructor defines that if `x ≤ y` and `Ordered (y :: xs)`, then `Ordered (x :: y :: xs)`. Then, define as a function the statement meaning for "the lists xs and ys have the same multiset" :

```
sameMultiset? : List ℕ → List ℕ → Set
sameMultiset? []        []        =  ⊤
sameMultiset? (x :: xs) (y :: ys) =  < implement it! >
sameMultiset? _         _         =  ⊥
```

(note: types are data, and this function returns a type). This code needs to express that each number `n` occurs in `xs` with the same multiplicity as in `ys`. It needs to return a *non-empty type* if and only if `xs` and `ys` have the same multiset. Further, program a sorting function, with the result including the sorted list and the correctness proof:

```
record Sort (xs : List ℕ) : Set where
                            field
                              resList       : List ℕ
                              ordProof      : Ordered resList
                              multisetProof : sameMultiset? xs resList
sort : (xs : List ℕ) → Sort xs
sort xs = ...
```

Here the correctness proof consists of the two last fields, which express the above definition of *what is a sorting map*. Spending some effort (a great effort for a newbie!), one can program this all so that

a) the part `resList` has the cost bound of $O(n*(log\ n))$ for n `= length xs`,
b) the same cost order bound has the orderedness proof `ordProof`,
c) the "multiset" proof cost bound is $O(n^2)$.
   This is because finding the multiplicity of `x` in `xs` needs `(length xs) - 1` comparisons (unless some particularly wise method is applied).

Also the program must include a proof for *termination.*

The approach is as follows. Apply the 'merge' method for sorting. The function `merge` merges two ordered lists into the list `zs`, and also returns a proof for

that `zs` is ordered. A proof is built recursively by the structure of the lists, and parallel-wise with evaluation of `zs`. Then, program sorting as splitting a list to halfs (by repeatedly moving a pair of elements from the list), sorting each half recursively, and applying `'merge'` to the sorted halfs. A proof for termination is included in this program by adding an additional counter value in the loop; in the form of certain concatenated lists, and by taking the tail of this counter at each step.
(So far, the author has programmed `'sort'` with skipping `multisetProof`).

But it is very difficult to program sorting so that all the above parts to have the cost bounded by $O(n*(log\, n))$. Namely, the point `multisetProof` is problematic. Even if we manage to do this, there still are possible more problematic examples.

Question aside: why do we mix in one function ordinary evaluation with a proof? Because if we split it into a function `f` for ordinary computation and to a proof function for `f`, this will most often lead to the two copies of a very similar code, where the second is a bit more complex than the first.

Return to the sorting example.

1. After the above program is type-checked — it is *verified*, together with the `multisetProof` part.

2. Suppose that a function `f` uses the result of `sort xs`:

```
f : List ℕ → List ℕ
f xs = g xs (resList res) (ordProof res) (multisetProof res)
  where
  res = sort xs
  open Sort
  g : (xs : List ℕ) → (ys : List ℕ) → Ordered ys →
                                       sameMultiset? xs ys → List ℕ
  g xs ys ord-ys sameMSet =  ...
```

Here `ys` and `ord-ys` cost $O(n*(log\, n))$ — if really used in `g`.
`sameMSet` ensures that `ys` has the same multiset as `xs` — this is another correctness condition for applying `g`.

Proofs are data, which constructors are defined in the user program. So, the function `g` may 'look' into the structure of the `sameMSet`, proof. And if `g` does analyze the `sameMSet` value, then `sameMSet` starts to really evaluate, and this may lead to the run-time 'explosion' of the $O(n^2)$ cost.

But in most cases there is no reason for `g` to analyze `sameMSet`. For correctness, there is sufficient only the fact of that `sameMSet` belongs to the needed type. And we can arrange a program (call it `sort'`) so that this fact occurs established by the type checker. Namely: **1)** program `sort1` which is like `sort` only skips the `multisetProof` part, **2)** program separately

`lemma : (xs : List ℕ) → sameMultiset? xs (list (sort1 xs))`,

3) set in the `sort'` result the first two fields from `sort1` and the third field as `multisetProof = lemma xs`.

For this design, `sameSet` costs nothing in `g` at the running time.
Currently I do not know of whether this rewriting to `sort'` is really necessary.

Probably, a similar reorganization (if needed) will solve this problem in other examples.

2.2 Objection Attempt 2. Solver Hierarchy

"It is difficult to write proofs in `Agda`".

The user needs to write proofs which look similar to ones given in the classical textbooks, for example, on algebra. The closer to this sample, the better.

By "writing a proof" I do *not* mean here *inventing a proof.*
This concerns only writing a proof in atomic details — after its main part has been invented and written in the form of a classical textbook. The matter is that even though such a humanly proof may be considered as "rigorous", it may be still technically difficult to "unwind" this proof into a formal proof for the `Agda` type checker
(note also that many lengthy "rigorous" proofs in classical books have typos and errors which make these proofs incorrect — which is not possible for a formal proof in a DT system).

Composing `Agda` proofs from atomic steps is difficult and also gives a large source code which is difficult to read. The style is like this:
"apply at this position transitivity of equality, at this position — congruence of `_*_`, associativity of `_+_`, here — commutativity of `_+_`", and so on, with providing the correspondent arguments.

The `EqReasoning` tool of Standard library actually automates the usage of an equality transitivity. This allows to write about 2 times shorter source proofs, which also are somewhat more readable.

Generally, this is nice that proof tools in `Agda` can be given in a library: just introduce an appropriate operator and implement in `Agda` the corresponding function.

Further, the `Ringsolver` tool of Standard library automatically provides a proof to any true equality `s` $\approx$ `t` in the free commutative algebra over `Integer`: $A = \mathbb{Z}[x_1, \ldots, x_n]$. This is the same as a polynomial algebra. Here the variables x_i correspond to the identifiers in the program which take part in the expressions `s` and `t`. By the function `'solve'`, each expression `s` and `t` is brought to the normal form, and these normal forms are compared. This gives a nice coding for many proofs. *Still writing/reading proofs remains unnaturally difficult.*

Algebraic 'Modulo' Solver. In the programming practice, the algebra `A` in which an equality `s` $\approx$ `t` needs to be proved most often is not a polynomial algebra `P`, but is $P/(e_1, \ldots, e_k)$ — a quotient of `P` by the given equations e_i. This is because the identifiers often are not independent: they satisfy some relations. For example, it is given that $x_1 + 2 * x_2 \approx x_3$ and $2 * x_2 - x_4 \approx x_1$, and one needs a proof for the equation `s` $\approx$ `t` *modulo* the above two equations.

If all the above equations are *linear*, the problem is reduced to solving a linear system over the domain of `Integer`. So, it is not difficult to implement a "modulo-linear" extension for `RingSolver`. Note that a correctness `Agda` proof for solving a system is not needed here. Because the found solution is an integer vector, which is then converted to the `Agda` proof, and it does not matter for the type checker of how this proof has been found.

The next possible level in the solver (prover) hierarchy is for the case when the equations are *non-linear and algebraic.* Again, there is known an algorithm for solving this problem: the Gröbner basis method [3] (there also is known its variant for the coefficient ring of `Integer`).

Also both methods are programmed in `Haskell` in the `DoCon` library [10].

However, we need to take in account that the latter algorithm may lead to an expensive computation to occur at the type-check time.

The next level in the solver hierarchy is for the case of *non-algebraic equations.* For this problem there is known the Knuth-Bendix method. Its variant [7] called "unfailing completion" is a semidecision procedure for this problem.

On practice, both the two latter methods will need an interactive proof in which the user gives some lemma equations during the type check.

The next level in the hierarchy is by the *interactive inductive prover.*

The more powerful provers are added to the library (to help the programmer and the type checker) the more real proof assistance will provide the `Agda` *proof assistant.*

Objection 2 will be removed by development of the prover library.

Objection 3 A shortly written and efficient algorithm may need a proof of a book having, say, 500 pages.

I think, this does not reject the constructive DT practice — due to the following reasons.

1) When mathematicians use this algorithm, they still refer to a proof in some existing book, and some of them do analyze this proof before programming or using this algorithm. Writing this book corresponds to writing the proof part in the corresponding `Agda` program. The proof check happens before running the algorithm, similar as it is in the classical computation.

2) In rare cases people apply an algorithm without *anyone* knowing of a rigorous proof for some its essential property. This often has sense.

And this corresponds to the `'postulate'` construct in an `Agda` program.

By this all, Objection 3 is removed.

Objection 4: Type Check Cost. An `Agda` program often has a pitfall for the type checker, due to *normalization* of type expressions. If the programmer has/uses a tool for restricting normalization, then proofs become more difficult to program. Because types depend on value expressions, and type normalization often helps to reach a proof. On the other hand, forgetting of possible normalization effect may lead to the type check "explosion", a great expense at the type check stage.

Objection 5. Cost Verification Most of the existing programs which have been type-checked in `Agda` are not still really verified!

Because the computation cost matters. Imagine that a source program has such a typo which keeps it type-checked but slows it down greatly. For example, the program may run 10 years instead of 1 second. Recall also that many works on algorithms have proofs for the evaluation cost bound formulae. It is natural to add these bounds to verification.

And this problem can be solved within the same DT paradigm. A program only needs to process recursively the corresponding cost proof data.

For example, return to the list sorting program. Suppose that we need to prove the upper bound $cost \leq n^2$ for its running time cost. And suppose that it is taken an admissible relative time measure: the number of the element comparisons applied. Add the argument value `cost : ℕ` to the loop body in the program (here it is better to have an n-ary positional arithmetics). Also add there a proof `p` for the current cost bound. And program a final cost proof recursively, similar as the orderedness proof, but with using the arithmetical laws for `_<_, _*_`. For example, after the list is halved into `xsL` and `xsR`, it holds by recursion

```
costL, costR ≤ (n/2)^2;  cost ≤ costL + costR + n/2 = 2*(n/2)^2 + n/2
```

— because `(merge xsL xsR)` costs not more than `n/2`. And it remains to program a proof for

```
2*(n/2)^2 + n/2  ≤ n^2 :  ...<==>  n/2 ≤ n^2/2  <==>  n ≤ n^2.
```

— coding such a proof in `Agda` is an usual exercise.

Objection 5 is removed.

Summary. So far, I find the two obstacles for the DT programming practice in `Agda`:

(1) not everything is clear about Objection 1,
(2) difficulties in composing a proof (after a rigorous humanly proof is ready),
(3) the danger of explosion by normalization at the type check stage.

The point (2) is a matter of developing provers (probably, as a part of the library). Probably, this is the main direction in making from `Agda` a tool for an adequate programming of mathematics.

The point (3) is not clear for me, so far. Some common approach is needed for a reliable control over the explosion by normalization at the type check stage.

3 Design for Algebra

The `DoCon` library variant for `Agda` is called `DoCon-A`.
This project is in its beginning, and it is rather experimental at the moment.
It is going to be open-source. So far, it is not stuck.

Below there follow considerations on some details of the project.

Are `Haskell` data classes needed in `Agda` ?
I have a preliminary impression that are not. Because

a) `Haskell` instances are difficult to resolve automatically,

b) an advanced algebra needs *overlapping multiparametric instances*, and this aggravates the problem,

c) dependent records of `Agda`, together with the constructs of `'open'`, `'using'`, `'renaming'`, and with hidden arguments, provide a flexible tool for modelling classes.

Below the word "class" applied in the context of `Agda` means a data class modelled by a dependent record of `Agda`.

Terminology: Classical Hierarchy

There is known the hierarchy of algebraic 'categories' given in the classical text-books on algebra: Semigroup, Group, Ring, and many others.

Here we call them the *classical (algebraic) hierarchy.*

3.1 Setoid

The user-defined equality `'=='` in `Haskell` does not necessarily satisfy the three equivalence laws, its safe implementation is on the programmer.

And with `Agda`, the classical hierarchy is naturally based on the `Setoid` class of Standard library, with its user-implemented equality $\_\approx\_$, and with the necessary *proof* implementation for the three equivalence laws, so that these laws are checked by the type checker.

About total functions. Note that proofs for the above equivalence laws (and for many other laws for programs) hardly ever have sense in presence of program breaks or non-terminating. For example, for functions `f, g :: Char -> Char`, is it true the implication `(f 'a' == g 'a') ==> (g 'a' == f 'a')` ?

In `Agda`, it does hold (for the relation $\_\approx\_$). Because 1) the programs for `f` and `g` are provided with a termination proof, and a function is total on its domain type (breaks are not possible), 2) an implementation for $\_\approx\_$ is provided with a proof for the three equivalence laws.

3.2 A Constant Operation Signature

For the *zero* and *unity* constants in an algebraic domain, the `DoCon` library (written in `Haskell`) uses the signature `: a -> a`.

This is forced by the feature of a *domain depending on a dynamic parameter* (as it is written in Section 1, an advanced algebra needs such). In `(zero s)`, `s` is a sample containing the domain parameters. For example, zero in a ring `V = Vector Integer xs` is different, depending on the length of the list `xs` giving the dimension of the vector. `Vec [0, 0]` and `Vec [0, 0, 0]` are zeroes of different domains, while they belong to the same type `Vector Integer`. The domain (inside a type) is defined by the parameters contained in a *sample element*, in this example this parameter is a list.

And `Agda` makes it possible a fully adequate representation:

```
unity? : (A : Setoid) → let C = Setoid.Carrier A in  Op₂ C → C → Set
unity? A _*_ e =  (x : Carrier) → ((e * x) ≈ x) × ((x * e) ≈ x)
```

```
                                                          where open Setoid A
Unity : (A : Setoid) → Op₂ $ Setoid.Carrier A → Set
Unity A _*_ =  ∃ (\ (e : Setoid.Carrier A) → unity? A _*_ e)
...
record Monoid (upSmg : UpSemigroup) : Set
  where
  upSemigroup = upSmg
  Smg         = UpSemigroup.semigroup upSmg
  private  open Semigroup Smg using (_≈_; _•_; ...)
                              renaming (Carrier to C; setoid to S; ...)
  field  unity : Unity S _•_

  ε : C
  ε = proj₁ unity
  ...
```

Here and below we skip the `Level` parameters in the code, because this language detail is not essential for this paper.

The `Monoid` classs is modelled by a record; it declares that `Monoid` is defined over a given `Semigroup`, and the operations $\_\cdot\_$ and `Carrier` (renamed to `C`) are imported from `Semigroup`. Its only `field` is the '`unity`' operation.

The traditional unity element is given by the constant ϵ, implemented as the first projection from `unity`. And the type `Unity` expresses the full notion of a unity in a semigroup. It means that applying `unity` finds an element `e` in `C` which satisfies the unity laws `(e · x) ≈ x`, `(x · e) ≈ x` for each `x : C`. And '`unity`' returns a pair: the unity element ϵ and *proofs* for the two correspondig equation laws. The library function ∃ in the definition of `Unity` has a constructive meaning.

3.3 DSet

The base for the `DoCon-A` hierarchy is the class

```
record DSet (decS : DecSetoid) : Set
  where
  decSetoid = decS
  private open DecSetoid decS using (setoid; Carrier; _≈_)
  ≈equiv = Setoid.isEquivalence setoid

  field  mbFiniteEnum : Maybe $ Dec $ hasFiniteEnumeration setoid
  ...
```

`DSet` is a set with a decidable equality relation $\_\approx\_$ on it.

Decidable equality. `DoCon-A` puts it so because an interesting computation can happen in a domain `D` only when there is given an algorithm for solving the equality relation on `D`. For example, having a commutative ring `R` and computing with the polynomial $f = (a - b) * x^2 + x$, where `a` and `b` are from `R`, how does one find the degree of f ? Is it 2 or 1 ? If this is not solved, then most of important computations are *not* possible in the domains related to `R`.

Return to `DSet`.

`mbFiniteEnum = just (yes fn)` means that the set has a finite enumeration presented by the data `fn`, together with a proof for surjectiveness of the enumeration list (with respect to $\_\approx\_$).

`mbFiniteEnum = just (no _)` means that the set is infinite.

`mbFiniteEnum = nothing` means "unknown".

Here is an example showing why this design is natural. Consider a *quotient group* $Q = G/H(g_1, g_2, g_3)$ of some non-commutative group of a complex nature by a normal subgroup `H` generated by the given three elements. Suppose that g_i are computed and are changed during evaluation. Depending on the current g_i values, the group `Q` may occur finite or not. The problem of deciding on its finiteness may be arbitrarily complex. This is why the value `nothing` is reserved to represent "unknown".

<u>Maybe–Dec approach</u> This approach, described above, is applied in the further class hierarchy. But it is not taken as total (otherwise one would have only a single class `DSet`, with thousands of maybe–dec operations – which does not look natural).

Thus solving a *division* equation in a semigroup may have arbitrary complexity depending on a dynamic domain parameter. There are many other examples.

3.4 Relation to the Standard Algebra Classes

Standard library for `Agda` (lib-0.7) is profoundly defined.
And `DoCon-A` uses a great part of it. As to the part of the proper classical algebraic hierarchy, `DoCon-A` defines by *new*: `DSet, Magma, Semigroup, Monoid`, and so on. Only a small part of the Standard library is out of `DoCon-A`:

`Semigroup, Monoid, CommutativeMonoid, ..., CommutativeRing`.

This is because `DoCon-A` is an *application* library aiming at the advanced algebraic problems having an algorithmic solution (described in varoius books and papers). For example: factoring polynomials over various appropriate commutative domains. An advanced algorithmic algebra requires certain additional operations for the corresponding classes. This is illustrated by the above example `DSet` and by the followng example with `Magma`.

Partial Operations. For example, the Integer ring $\mathbb{Z}$ has partial division and inversion: `div 4 2 --> just 2;` `div 5 2 --> nothing.`
Respectively, `Ring` and `Semigroup` need to have the operation for a partial division. It has many differently defined instances, and in some of this instances division occurs total (like it is in `Group`). The latter case is expressed by applying `(just? r)` $\equiv$ `true`, where `r` is the result of a partial division. Due to all this partial division is defined *conditionally* in `Magma` (a superclass for `Semigroup`):

```
record Magma (upDS : UpDSet) : Set
  where
  upDSet = upDS
  private  dS = UpDSet.dSet upDS
```

```
open DSet dS using (≈equiv; _≈_; decSetoid)
             renaming (Carrier to C; setoid to S)

open IsEquivalence ≈equiv using () renaming (refl to ≈refl)
  field
    _•_           : Op₂ C
    •cong         : _•_ Preserves₂ _≈_ → _≈_ → _≈_
    mbCommutative : Maybe $ Dec $ Commutative S _•_
    divRightMb    : (x y : C) → Maybe $ Dec $ RightQuotient S _•_ x y

  •cong₁ : {y : C} → (\x → x • y) Preserves _≈_ → _≈_
  •cong₁ x=x' =  •cong x=x' ≈refl
```

Here `divRightMb` returns a right-hand quotient for `x/y` in the maybe–dec format. In the just–yes case, the result also contains a proof for the equation defining of what is a quotient.

`Magma` is a set with a binary operation, which operation in congruent by the underlying equality (and with the two more operations specific for `DoCon-A`).

`updS` (for `DSet`) is an argument for the `Magma` class, because there are often needed different magmae (or semigroups) with the same `DSet`. For example, `+Magma of Integer` and `*Magma of Integer`.

Argument domain approach. The above example reflects the generic approach of the argument domain for a class. If we move `updS` from arguments and make it a field in the above record, then we loose the ability to express that two magmae are over the same `DSet`.

A similar consideration is applied to the further class hierarchy.

Again, commutativity (`mbCommutative`) is under maybe-dec, because it is not always easy for an algorithm to decide. If it is solved positively, the result (of the type `Commutative S _·_`) contains the corresponding proof.

Note that the `Magma` record also contains the lemma proof `·cong₁`, which is not a record field, but has an implementation relying on the field of `·cong` as on an axiom.

3.5 Up-domains

We have pointed earlier that most algebraic classes need some domains as arguments. `Magma` is over `DSet`, `Semigroup` is over `Magma`, `Ring` is over `CommutativeGroup` and `Semigroup`. This approach with the domain arguments will lead, for example, to that in the code fragment `R : Ring <args>` it will be necessary to set many agruments in the place of `<args>`. Our so-called up-domain approach solves this technical problem. Besides `Magma`, `DoCon-A` also declares its ‘up’ version:

```
record UpMagma : Set where field upDSet : UpDSet
                                 magma  : Magma upDSet
                             open UpDSet upDSet public
                             open Magma magma public using ...
```

Actually `UpMagma` is `Magma` — with the value for the agrument domain for `Magma` provided in the `upDSet` field. Similarly, there is `Group` and `UpGroup`, ..., `Ring` and `UpRing`, and so on. This leads to that the corresponding member in this class hierarchy needs 1-2 arguments instead of many, and on the other hand, it is easy to express the situation when two domains have a common argument domain. For example, to define a *linear* map `f : U → V` for the vector spaces, we need these spaces to be over a same `Field K`. And it is sufficient to provide a signature of kind

```
f : (upF : upField) → (upGU upGV : UpCommutativeGroup) →
    (U : VectorSpace upF upGU) → (V : VectorSpace upF upGV) ...
```

Here the same `Field K` will be extracted from `upF`, and different additive vector groups will be extracted from `upGV` and `upGV` respectively.

3.6 Further Algebra Class Hierarchy

This is `CommutativeSemigroup, Monoid, CommutativeMonoid, Group, CommutativeGroup, Ringoid, Ring, RingWithOne, CommutativeRing, IntegralRing, LinearSolvableRing` (a generalization for a ring with Gröbner bases), `GCDRing` (a ring where the greatest common divisor has sense), `FactorizationRing, EuclideanRing, Field, LeftModule` (over a ring) — and some others need to join.

3.7 Sub-domains

A subdomain is modelled in `DoCon` (in `Haskell`) by a symbolic represenattion, by coding. For example, an *ideal* in a `Ring` is represented as something like a data `(Ideal generatorList <otherAttributes>)`. The membership to a subdomain is not a matter of the compiler, but it is on the `DoCon` functions, and on the user functions.

With `Agda`, `DoCon-A` applies a fully adequate approach: everything is expressed by dependent types and is subjected to the type checker. An ideal also has a subring and a ring instances in it, an additive subgroup, and so on. Subdomains start with

```
record DecSubset (A : Set) (member? : A → Bool) :  Set where
       constructor _cond_
       field  repr        : A
              member-repr : member? repr ≡ true
```

— a decidable subset defined by a *membership predicate.*
Example: `Even = Subset ℕ even?;   (6 cond (even? 6)) : Even.`

`Submagma` is expressed as

```
record Submagma (upM : UpMagma)
                (eSubDSet : SubDSet $ UpMagma.updSet upM) :  Set
       where  ...
```

It is defined by a subset `S' (SubDSet)` and by the property `closed'` of `S'` being closed under `_·_`. It contains the field `submagma` representing the submagma as a `Magma` of the given subset, and certain other attributes, like imbedding to the embracing magma. In a similar manner there are defined `Subsemigroup, Subgroup, ..., Subring, Ideal`.

Defining a subdomain only by a membership function is not enough for practice. We add the description by a finite set of generators (for subsemigroup, ..., ideal). Computing in the *residue ring* of `R` by an ideal `I` needs `R` and `I` supplied with certain additional operations. And so on.

4 Conclusion

The dependently typed paradigm has proved as promising in computer algebra. It needs further practical investigation.

References

1. Agda. A dependently typed functional programming language and its system. `http://wiki.portal.chalmers.se/agda/pmwiki.php`
2. Augustsson, L.: Cayenne — a language with dependent types. In: International Conference on Functional Programming (ICFP'98). ACM Press, 1998.
3. Buchberger, B.: Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. CAMP. Publ. No.83–29.0 November 1983
4. The Coq Proof Assistant. `http://coq.inria.fr`
5. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`
6. Haskell 2010: A Non-strict, Purely Functional Language. Report of 2010. `http://www.haskell.org`
7. Hsiang, J., Rusinowitch, M.: On word problems in equational theories. In Th. Ottman (ed.), Fourteenth International Conference on Automata, Languages and Programming, Karlsruhe, West Germany, July 1987, LNCS, vol. 267, pp. 54–71, Springer Verlag (1987)
8. Per Martin-Löef: Intuitionistic Type Theory Bibliopolis. ISBN 88-7088-105-9 (1984).
9. Mechveliani, S. D.: Computer algebra with Haskell: applying functional-categorial-'lazy' programming. In: International Workshop CAAP-2001, Dubna, Russia, pp. 203–211 (2001) `http://compalg.jinr.ru/Confs/CAAP_2001/Final/proceedings/proceed.pdf`
10. Mechveliani, S. D.: DoCon. The Algebraic Domain Constructor. A program source and a manual. Pereslavl-Zalessky, Russia. `http://www.botik.ru/pub/local/Mechveliani/docon/`
11. Norell, U., Chapman, J.: Dependently Typed Programming in Agda. `http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`