Provable programming of algebra: particular points, examples.

Sergei D. Meshveliani

Abstract. It is discussed an experience in provable programming of a computer algebra library with using a purely functional language with dependent types (`Agda`). There are given several examples illustrating particular points of implementing the approach of constructive mathematics.

Keywords. constructive mathematics, computer algebra, dependent types.

1. Introduction

Applying the approach of constructive mathematics [2] [1] and technology of purely functional programming with *dependent types* [5] makes it a fully adequate approach to programming computation in algebra, in mathematics. In particular [3] [4] this **1)** solves completely the problem of representing an algebraic domain depending on a dynamic parameter, **2)** makes mathematical definitions and formal proofs an explicit part of a program, a part understood by the compiler and automatically checked before the running time.

So far, we use `Agda` as a language with dependent types.

This thesis describes in examples some of interesting features of the constructive-provable approach to programming algebra.

2. Example with termination proof

The *type checker* needs to verify termination for each function given in the program, in order to follow constructive mathematics. The main tool for this is finding an argument which is decreased in a certain syntactic ordering when the function is applied recursively. For example, apply the unary coding for natural numbers, with the constructor `suc` for the next number. Consider the program

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

```
0       + n = n
(suc m) + n = suc (m + n)
```

for summing natural numbers. The type checker decides that it is terminating, because in the second line the left argument term 'm' for _+_ on the right hand side is syntactically smaller than the argument (suc m) on the left hand side. For more complex functions, the programmer is often supposed to help the type checker by introducing the counter expression. The counter is syntactically decreased with each recursive call. And the program needs to specify the result when the counter turns zero. Consider the example for a provable program:

for any prime natural p *find the next prime.*

In reality, we tested the sieve method. But for this paper, let us consider the simplest algorithm of searching-through. The primality notion for $\mathbb{N}$ is defined in Agda as the function

```
IsPrime : ℕ → Set
IsPrime p =  p ≢ 1  ×  (∀ {m n} → (m * n ≡ p) → m ≡ 1 ⊎ n ≡ 1)
```

that *returns a type*. This type has a value in it when the argument p is prime. It expresses the property "if m * n ≡ p, then m ≡ 1 or n ≡ 1".

Assume the following simplest constructs. The algorithm _|?_ for deciding divisibility is defined via division with remainder. The algorithm prime? for deciding primality of n is defined by searching through all 1 < m < n with applying m |? n. The algorithm firstFactor>1 finds for each n > 1 the first m > 1 that divides n. This is done by searching through. Also it is proved the statement
n > 1 → IsPrime (firstFactor>1 n). Now, the function

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

needs to return q which is the first prime after p (and the corresponding witnesses). Program it as search-through: test (prime? n) for n = 1+p, 2+p, ..., until is finds a prime. To provide a termination proof, let us give an expression for *any* prime b = bound p such that p < b. Then, add the counter expression cnt = b - n to the search loop. The counter is decreased each time when n steps from n to suc n. If a prime is not found earlier, and it turns cnt = 0, then n ≡ b occurs the needed prime. With this, the type checker verifies termination.

And for b = bound p we need any expression such that the properties p < b and IsPrime b have an easy constructive proof. Choose this:

```
bound p = let  p₀, p₁, ..., p  =  all primes up to  p
                                  -- found by repeatedly applying 'prime?'
               a =  p₀ * p₁ * ... * p
          in
          firstFactor>1 (1 + a)
```

Then, a constructive proof for that (bound p) is a prime greater than p is not difficult to provide. With this, the program is verified. But it performs in a strange way at run-time: (nextPrime 31) hangs for a very long time!

The reason for this is that the condition b - n =? 0 is very expensive to check at run-time (see the algorithm for b). Its check is needed only to provide (statically) a termination proof, it is not needed at run-time. But there is no way to explain this to Agda, and this check gets into the executable program.

Improvement: apply the Bertrand – Chebyshev estimation —
"there exists a prime between `p` and `2*p`".

The counter of `2*p - n` is compared fast to zero. But a proof for this bound is large and complex! So: one bound is easy to prove but expensive to compute at run-time, another bound is computed fast but is complex to prove. What has one to put into the `Agda` program? The way out for `(nextPrime p)` is as follows.

Search-II:

first search before `bound2 = 2*p`. If the needed `q` is found, then stop.

Otherwise, search by new from `1+p` up to `(bound p)`.

This program 1) is verified, including termination, 2) has a fast comparison in the search loop. This is because the part of 'Otherwise' will never be performed at run-time.

But what is a way out for the case when there is proved an 'expensive' bound like above, and is not even known of any better bound?

The following solution is sufficient. In Search-II, put for `bound2` an unfeasible number — such one that will never be reached in practice in the above loop (I take this solution from the message by Ulf Norell).

In fact, this trick with unfeasible bound partly replaces the Markov's principle in constructive mathematics. This principle [2] allows a proof by contradiction for a termination proof, and it cannot be implemented in `Agda` without using the `'postulate'` construct.

3. Refuting the two prejudices

Prejudice 1:
"Proof by contradiction is not possible in constructive mathematics".

In fact: *it is possible* — when the relation has a decision algorithm.
Example: in most domains in computer algebra the equality relation has a decision algorithm `_=?_`. Respectively, a program of the kind

```
case x =? y of \ { (yes x≈y) → ...;  (no x≉y) → ... }
```

actually applies the *excluded third* law to this relation.

Prejudice 2: "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

$$\text{for all } n\ (n \le n).$$

for natural numbers. The relation `_≤_` is defined on $\mathbb{N}$ so that a witness for it can be built only with applying the two data constructors (axioms): `z≤n` — "`0 ≤ n` for all `n`", and `s≤s` — "if `m ≤ n`, then `suc m ≤ suc n`".
(Syntax: `z≤n, s≤s` are function *names*, as they are written without blanks).

For example: `(s≤s (s≤s z≤n))` is a proof for `2 ≤ 5`.

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

Write the corresponding proof in `Agda`:

```
theorem :  ∀ n → n ≤ n
theorem 0       =  z≤n
theorem (suc n) =  s≤s (theorem n)
```

For each `n : ℕ` the function `theorem` returns a value in the type `n ≤ n`, that is the corresponding witness. The second pattern applies the function `theorem` recursively. This provides a *proof in the two meanings*.
(1) At the run-time, `(theorem n)` yields a proof for `n ≤ 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 ≤ 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.

References

[1] Per Martin-Löef, *Intuitionistic Type Theory*. Bibliopolis. ISBN 88-7088-105-9, 1984.

[2] 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, 8–14.

[3] 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), pp. 27–50, available at
`http://psta.psiras.ru/read/psta2014_3_27-50.pdf`

[4] 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), available at
`http://psta.psiras.ru/read/psta2015_4_313-340.pdf`

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

Sergei D. Meshveliani
Program Systems Institute of Russian Academy of sciences, Pereslavl-Zalessky, Russia
e-mail: `mechvel@botik.ru`