Towards Standard Environments for Formalizing Mathematics

Adam Grabowski[1] and Christoph Schwarzweller[2]

[1] Institute of Informatics, University of Białystok
ul. Akademicka 2, 15-267 Białystok, Poland
adam@math.uwb.edu.pl
[2] Department of Computer Science, University of Gdańsk
ul. Wita Stwosza 57, 80-952 Gdańsk, Poland
schwarzw@inf.univ.gda.pl

Abstract. Though more and more advanced theorems have been formalized in proof systems their presentation still lacks the elegance of mathematical writing. The reason is that proof systems have to state much more details – a large number of which is usually omitted by mathematicians. In this paper we argue that proof languages should be improved into this direction to make proof systems more attractive and usable – the ultimate goal of course being a like-on-paper presentation. We show that using advanced Mizar typing techniques we already have the ability of formalizing pretty close to mathematical paper style. Consequently users of proof systems should be supplied with environments providing and automating these techniques, so that they can easily benefit from these.

1 Introduction

Interactive reasoning aims at developing methods and systems to be used to formalize – state and prove – mathematical theorems in a comfortable way. The ultimate dream is a system containing all mathematical knowledge in which mathematicians develop and prove new theories and theorems. Though more and more advanced pieces of mathematical knowledge are being formalized, we are still far from this dream – in particular few mathematicians even notice proof systems.[3] Formalizing mathematics more or less still is a matter of computer scientists.

In our opinion the main reason is the clash between how mathematicians and proof systems work: Any proof system by nature is based on logical rigour to ensure correctness of formalization. Consequently such systems state theorems more or less as logical formulae and use a logical calculus doing inferences to prove them. Mathematicians, however, do not use logic or logical symbols in

[3] The most prominent exception is Thomas Hales' Flyspeck project [Fly14].

the strong sense of proof systems. They argue rather intuitively assuming that their arguments can be easily transformed into such a logical reasoning. From this stem reservations against using proof systems like “Theorems are hard to read”, “Too much obvious facts has to be explicitly considered”, or “Applying theorems is too elaborated”.

To illustrate the above we consider the term $n - 1$ with $n \in \mathbb{N}$ as an easy example. In some situations – to apply a theorem or to use $n-1$ as an argument of a function – $n-1$ has to be a natural number. This is of course obvious if $n \geqslant 1$ (or $n \neq 0$), so mathematicians do not care about. Proof systems have to be much more formal: One has to prove that in this particular situation $n-1 \in \mathbb{N}$. This is of course pretty easy and can for example be done by changing the type of $n-1$ to $\mathbb{N}$, using the minus function $n \dot{-} 1$ or by generating some proof obligation. Somewhat more involved examples would be $(p-1)/2 \in \mathbb{N}$, if $p \neq 2$ is a prime or that $(-1)^n = -1$, if n is odd.

Though there have been efforts to overcome these shortcomings, we claim that in proof systems this kind of mathematical obviousness should be more strengthened: Proofs as those in the above example must be invisible for users, that is automatically identified and conducted. In this paper we show that Mizar’s attributed types [Miz14,Ban03] can be used to do so: Providing a number of so-called registrations and redefinitions – stored and made available to users in a special environment – automates reasoning as sketched in the above example and therefore allows for a much more mathematicians-like handling of mathematical knowledge. More concrete, we present examples from number theory – which in particular includes theorems as mentioned above – and deal with instantiation of algebraic structures.

2 Pocklington’s Theorem

Pocklington’s criterium is a number theoretical result providing a sufficient condition for (large) numbers to be prime. It may be worth mentioning that in the original work [Poc14] there is no precisely stated theorem. In the literature one therefore finds a number of different variants. One version (from [BM92]) reads as follows.

> Let s be a positive divisor of $n-1$, $s > \sqrt{n}$. Suppose there is an integer a satisfying:
>
> $$a^{n-1} \equiv 1 \pmod{n}$$
> $$\gcd(a^{(n-1)/q} - 1, n) = 1$$
>
> for each prime q dividing s. Then n is prime.

One can find several formalizations of Pocklington’s criterium none of which, however, resembles completely the mathematical formulation. In Mizar [Ric06] we find for example the following.

```
for n,f,d,n1,a,q being Element of NAT
st n-1 = q|^n1 * d & q|^n1 > d & d > 0 & q is prime &
   a|^(n-'1) mod n = 1 & (a|^((n-'1) div q)-'1) gcd n = 1
holds n is prime;
```

As we see the minus function `-'` and division with remainder though not being part of the theorem are used. Furthermore besides `mod` function and `prime` the formalization does not use no number theoretical notation, even divisibility is expressed implicitly.

The formalization found in [Cha08] adds `coprime` as a number theoretical notation in this way substituting the `gcd` function. Note, however, that divisibility is expressed once explicitly using predicate | and once implicitly.

$$n \geqslant 2 \wedge n-1 = q \cdot r \wedge n \leqslant q^2 \wedge a^{n-1} \equiv 1 \ (mod \ n)$$
$$\wedge (\forall p.\texttt{prime}\ p \wedge p|q \longrightarrow \texttt{coprime}(a^{\frac{n-1}{q}} - 1)\ n) \longrightarrow \texttt{prime}\ n.$$

In Coq [Coq14] a somewhat different version has been formalized using a partial factorization of $n-1$. Therefore lists of natural numbers have been used. Congruence of numbers is here expressed using the `Mod`-function, and `S` and `pred` denote $n+1$ and $n \dot{-} 1$, respectively.

$$\begin{array}{l}
\forall(\texttt{n q m} \ : \texttt{nat})\ (\texttt{a} : \texttt{Z})\ (\texttt{qlist} : \texttt{natlist}),\\
\texttt{n} > 1 \rightarrow\\
\texttt{n} = \texttt{S}\ (\texttt{q} \times \texttt{m}) \rightarrow\\
\texttt{q} = \texttt{product qlist} \rightarrow\\
\texttt{allPrime qlist} \rightarrow\\
\texttt{Mod}\ (\texttt{Exp a}\ (\texttt{pred n}))\ 1\ \texttt{n} \rightarrow\\
\texttt{allLinCombMod a n m qlist} \rightarrow \texttt{n} \leqslant \texttt{q} \times \texttt{q} \rightarrow \texttt{Prime n}.
\end{array}$$

Though of course correct – and also more or less well-readable – all these formalizations rather present themselves as an expression having been proved in a formal system than as a well-formulated mathematical theorem. In the rest of this section we will show how preparing a number theoretic environment allows for the following Mizar formulation of Pocklington's theorem.

```
for n being 2_greater natural number,
    s being non trivial Divisor of n-1 st s > sqrt(n) &
  ex a being natural number
  st a|^(n-1),1 are_congruent_mod n &
     for q being PrimeDivisor of s holds a|^((n-1)/q) - 1 gcd n = 1
holds n is prime;
```

2.1 Preparing Proper Mathematical Notation

The first step is obvious. We have to introduce definitions resembling the mathematical objects of concern: `Divisor`, `PrimeDivisor`, `are_congruent_mod`, and so on. Some of them were already available in the Mizar Mathematical Library, some we had to introduce by ourselves. How to do this has been described for example in [GKN10] and is therefore omitted. However, we also introduced the $<$-relation as a Mizar adjective `_greater` by the following attribute definition.

```
definition
  let n,x be natural number;
  attr x is n_greater means x > n;
end;
```

This at first sight seems to be an unecessary repetition. However, Mizar adjectives can be used in so-called cluster registrations to automatically extend and enrich objects' types. For example, the fact that primes $p \neq 2$ are odd can now be formulated – and proved – not only as a theorem, but also as a cluster registration:

```
registration
  cluster 2_greater -> odd for PrimeNumber;
end;
```

As a consequence having `p` of type `2_greater PrimeNumber` Mizar automatically adds the adjective `odd` to the type of `p` in this way adding hidden information about mathematical objects – that mathematicians use implicitly. In section 2.2 from this then will – also automatically – follow that `(p-1)/2` for such `p` is actually a natural number. To give another example here, the existence of an arbitrary large prime number can be guaranteed by the following registration.

```
registration
  let n be natural number;
  cluster n_greater PrimeNumber;
end;
```

Now, if necessary, the user can declare an arbitrary large prime number by just writing `let p be 12345_greater PrimeNumber;` or even more generally by `let n be natural number; let p be n_greater PrimeNumber;`. Its existence is guaranteed by the above registration and the fact that `p` is greater than `12345` or `n` respectively can be used in the following without any proving or referencing.

2.2 Automatically Adapting Types

The cluster mechanism of adding adjectives to an object's type from the last subsection can be used to automatically adapt tpyes in particular situations. In this way users – like mathematicians – do not have to deal explicitly with changing and adapting types to apply functors or theorems.

To deal with the easy example from the introduction first, if $n \in \mathbb{N}$ is not equal to 0 the type of $n - 1$ of course can be changed to natural number. To do this automatically, we identify properties – given by adjectives – ensuring that $n - 1 \in \mathbb{N}$ and formulate corresponding registrations, such as for example

```
registration
  let n be non zero natural number;
  cluster n-1 -> natural for non zero;
end;

registration
  let m be natural number;
  cluster m_greater -> non zero for natural number;
end;
```

Note here that registrations do not stand alone, but are applied in a iterative matter.[4] As a consequence the type of $n-1$ now happens to be `natural number` not only if n is `non zero`, but also if n is `m_greater` for an arbitrary natural number m.

We end this section by illustrating how the use of adjectives and cluster registrations allows to avoid additional helper functions such as minus and division with remainder to formulate Pocklington's theorem. Having the following registration

```
registration
  let n be odd natural number;
  cluster (n-1)/2 -> natural;
end;
```

then, if `p` is of type `2_greater PrimeNumber` the type of `(p-1)/2` is not just `real number` as given by the type of the division functor `/`. Together with the registrations from section 2.1 both adjectives `odd` and then `natural` are added to the type of `(p-1)/2`. Hence its type in particular is `natural number` and `(p-1)/2` is therefore accepted as the argument of a function requiring natural numbers. Note that once the registration has been introduced, no proof obligation for the user shows up, all that's necessary has – and must have – been proved in the cluster registration. Using the earlier introduced type `Divisor` the following

```
registration
  let n be natural number;
  let q be Divisor of n;
  cluster n/q -> natural;
end;
```

now is an easy generalization of the former case – `q` = `2` – changing the type of a quotient to natural number, as necessary in the formulation of Pocklington's theorem. Note again that the type of `n/q` is automatically enriched with adjective `natural`, if `n` and `q` have the attributed types mentioned in the registration.

[4] Actually Mizar rounds up an object's type by adding all adjectives from clusters available in the environment, see [Ban03].

3 Abstract Mathematical Structures and Intstantiations

Another main topic is moving between mathematical structures: Mathematical proofs receive their elegance from noting that a given domain constitutes a special structure and applying theorems from it. Here both jumping to completely different structures as well as inheriting from more general structures is of concern. In proof systems, however, this goes along with a type coercion. The type of an element of a ring is different from the one of a real number, of an element of a group or a topological space. Much effort has been spent to ease users of proof systems to move between and to apply theorems from different structures, see e.g. [GPWZ02], [Bal14], [RST01], [Sch07].

Here we deal with another topic connected with inheriting from general structures: Functions and properties defined in a general structure are to be refined or extended in a more conrete one. As a running example we consider greatest common divisors in different domains. The greatest common divisor and a number of its basic properties can be defined for arbitrary gcd domains. Note, however, that one cannot define a gcd function, just because in general the gcd is not unique. In gcd domains we therefore end up with a type `a_gcd`:[5]

```
definition
  let L be non empty multMagma;
  let x,y,z be Element of L;
  attr z is x,y-gcd means
    z divides x & z divides y &
    for r being Element of L
                st r divides x & r divides y holds r divides z;
end;

definition
  let L be gcdDomain;
  let x,y be Element of L;
  mode a_gcd of x,y is x,y-gcd Element of L;
end;
```

In more concrete gcd domains – so-called instantiations – such as the ring of integers or polynomial rings the notion of a gcd now is adopted – actually changed into a gcd function – by just saying that the gcd is greater than 0 or the gcd is monic, respectively. However, these additional properties apply only to objects of the more concrete type – `Integer` and `Polynomial` – whereas `a_gcd` expects arguments of the more general type `Element of L`, where `L` is a `gcdDomain`. To easily adopt mathematical refining techniques we need a way to – automatically – identify these types.

[5] One can of course also define the set of gcds for given x and y, but we found it more convenient to use Mizar types here.

3.1 Preparing the Instantiation

Instantiations of abstract structures are defined by gluing together the corresponding objects and operations in the appropriate structure, in our example `doubleLoopStr`. So the ring of polynomials with coefficients from a structure `L` can be defined by

```
definition
  let L be Ring;
  func Polynom-Ring L -> strict non empty doubleLoopStr equals
    doubleLoopStr(#POLYS,polyadd(L),polymult(L),1_.(L),0_.(L)#);
end;
```

where `POLYS` is the set of objects with type `Polynomial of L`. So we are left with two different types `Polynomial of L` and `Element of the carrier of Polynom-Ring L`, the latter one being the type of the – abstract – ring elements. As a consequence special properties of polynomials can be only defined for the concrete type `Polynomial of L`, such as for example an adjective `monic` . Even after its definition, `monic` is not available for objects of type `Element of the carrier of Polynom-Ring L`:

```
definition
  let L be Ring,
      p be Polynomial of L;
  attr p is monic means Leading-Coefficient p = 1.L:
end;

now let L be Ring;
  let p be Element of the carrier of Polynom-Ring L;
  p is monic;
::>    *106: Unknown attribute
    ...
end;
```

This is unsatisfying not only because it is obvious that `p` in this example is actually a polynomial, but also because it prevents the combination of `monic` with the former defined type `a_gcd`. The solution is to automatically cast abstract types into concrete ones, here `Element of the carrier of Polynom-Ring L` into `Polynomial of L`. Again attributes – and a so-called redefinition – allow both to describe such situations and to enhance Mizar type checking: First an attribute `polynomial-membered` describes sets containing only elements of type `Polynomials of L`. Then for such sets the type of its elements can be redefined into `Polynmial of L` – because the attribute `polynomial-membered` ensures that this cast is possible.[6]

[6] In fact exactly this has to be proved in the redefinition.

```
definition
  let L be non empty ZeroStr;
  let X be set;
  attr X is L-polynomial-membered means
    for p be set st p in X holds p is Polynomial of L;
end;

definition
  let L be Ring;
  let X be non empty L-polynomial-membered set;
  redefine mode Element of X -> Polynomial of L;
end;
```

All that remains now is stating – and proving – a cluster saying that the type `the carrier of Polynom-Ring L` can automatically be enriched with the adjective `polynomial-membered`

```
registration
  let L be Ring;
  cluster the carrier of Polynom-Ring L -> L-polynomial-membered;
end;
```

and objects of type `Element of the carrier of Polynom-Ring L` are automatically identified as objects also having the concrete type `Polynomial of L`. Therefore notions defined for `Polynomial of L` – such as for example `monic` – are also available for objects of type `Element of the carrier of Polynom-Ring L`.

3.2 Extending Definitions in the Instantiation

Working in an environment containing the clusters and redefinitions from the last section users can now extend and combine properties defined for different types – abstract ring elements and concrete polynomials – according to their needs. First – if not already provided in the standard environment – one has to ensure that the instantiation establishes the abstract structure, here that `Polynom-Ring L` is a gcd domain.

```
registration
  let L be Field;
  cluster Polynom-Ring L -> Euclidian;
end;
```

Then all is set: Both abstract notions from `gcdDomain` and concrete ones for `Polynomials` are available and can be easily used. The definition for polynomial gcd function, for example, is now just combining notions `a_gcd` and `monic`.

```
definition
  let L be Field;
  let p,q be Element of the carrier of Polynom-Ring L;
  func p gcd q -> Element of the carrier of Polynom-Ring L means
    it is a_gcd of p,q & it is monic;
end;
```

Please note again, that notion `monic` has been introduced for objects of type `Polynomial of L`, whereas notion `a_gcd` for objects of type `Element of the carrier of Polynom-Ring L`. To nicely complete the development of polynomial gcds one should in addition provide the following registration enriching the type of the just defined gcd function – to make available properties of polynomial gcds without the need of referencing its definition.[7]

```
registration
  let L be Field;
  let p,q be Element of the carrier of Polynom-Ring L;
  cluster p gcd q -> monic p,q-gcd;
end;
```

4 Conclusion and Further Development

We have seen how thorough preparation of Mizar environments using registrations and redefinitions to manipulate mathematical objects' types not only improves working in a special mathematical theory, but also enables automatic use of hidden information – information that is used implicitly but is not stated by mathematicians. It is this kind of obvious knowledge and inferences that proof systems must enable in order to attract more users. We claim that further development of the presented techniques will lead to both more convenience in formalizing mathematics and more recognition of proof systems by mathematicians.

In this context we want to discuss briefly two further items that seem important to us. Firstly, a lot of theorems mathematicians do not mention in proofs can actually be presented as term reductions, just because they describe equalities, such as for example $(-1)^n = -1$, if n is odd, $v - v = 0$, $a \wedge b = a$, if $a \leqslant b$ or $x \cup \varnothing = x$. Of course it depends on the proof assistant to which extend such equalities/reductions are automatically applied. Mizar, however, provides a language construct similar to clustering that enables users enriching the proof process by particular reductions [Kor13], so for example

```
registration
  let n be odd natural number;
  reduce (-1)|^n to -1;
end;
```

[7] There is a relatively new Mizar environment directive – `expansion` – that also serves for autmatically applying definitions.

After such a registration the Mizar prover automatically identifies the left term with the term on the right side, so that – even depending on the object's type – equalities are automatically performed and accepted.

```
now let n be odd natural number;
  5 * (-1)|^n = -5;
  ...
end;
```

Unfortunately not all examples from above can be registered this way, because at the moment reductions must be to proper subterms.

The second point concerns the use of ellipses, a constituent mathematicians use to eliminate logical formulae describing finite sequences. In [CO01] we find, for example, Pocklington's theorem as follows:

> Let $n \in \mathbb{N}$, $n > 1$ with $n - 1 = q \cdot m$ such that $q = q_1 \cdots q_t$ for certain primes $q_1, \ldots q_t$. Suppose that $a \in \mathbb{Z}$ satisfies $a^{n-1} = 1 \pmod{n}$ and $\gcd(a^{\frac{n-1}{q_i}} - 1, n) = 1$ for all $i = 1, \ldots t$. If $q > \sqrt{n}$, then n is a prime.

The Coq version of Pocklington's theorem mentioned in section 2 actually formalizes this theorem and therefore uses lists of natural numbers. Mizar already offers the use of ellipses, but only if the underlying formula is existential [Kor12], like for example in the following theorem.

```
  for n being non zero natural number,
      x being integer number holds
  x,0 are_congruent_mod n or ... or x,(n-1) are_congruent_mod n;
```

This, however, unfortunately does not allow to formulate Pocklington's theorem with the use of ellipses. In particular the use of ellipses for indexed variables is necessary here.

Summarizing, proof assistants should be capable of automatically identifying and performing obvious mathematical arguments and shortcuts, that is arguments left out by working mathematicians. In our opinion the way to do so is not strengthening the proof assistant's inference system, but making its proof languages more flexible and adaptable by using language constructs like those presented in the paper: They allow to explicitly state theorems behind mathematicians' obvious arguments and then enrich the proof process by automatically applying them. In this way users – or developers by providing standard environments for formalizing mathematics – can adapt the proof process according to their particular needs.

References

[Bal14] C. Ballarin, Locales: A Module System for Mathematical Theories; Journal of Automated Reasoning, vol. 52(2), pp. 123-153, 2014.

[Ban03] G. Bancerek, On the Structure of Mizar Types; Electronic Notes inTheoretical Computer Science, vol. 85(7), pp. 69-85, 2003.

[BM92] J. Buchmann, V. Müller, Primality Testing; available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.1952`

[Cha08] A. Chaieb, Automated Methods for Formal Proofs in Simple Arithmetics and Algebra; Dissertation, TU München, Germany, 2008.

[CO01] O. Caprotti, M. Oostdijk, Formal and Efficient Primality Proofs by Use of Computer Algebra Oracles; Journal of Symbolic Computation vol. 32(1/2), pp. 55-70, 2001.

[Coq14] The Coq Proof Assistant; `http://coq.inria.fr`.

[Fly14] The Flyspeck Project; `http://code.google.com/p/flyspeck`.

[GKN10] A. Grabowski, A. Korniłowicz, A. Naumowicz, Mizar in a Nutshell; Journal of Formalized Reasoning, vol 3(2), pp. 153-245, 2010.

[GPWZ02] H. Geuvers, R. Pollack, F. Wiedijk, J. Zwanenburg, A Constructive Algebraic Hierachy in Coq; Journal of Symbolic Computation, vol . 34(4), pp. 271-286, 2002.

[GS10] A. Grabowski, C. Schwarzweller, On Duplication in Mathematical Repositories; in S. Autexier et.al. (eds.), Intelligent Computer Mathematics, LNCS 6167, pp. 427-439, 2010.

[Kor09] A. Korniłowicz, How to define Terms in Mizar Effectively?; Studies in Logic, Grammar and Rhetoric, vol. 18(31), pp. 67-77, 2009.

[Kor12] A. Korniłowicz, Tentative Experiments with Ellipsis in Mizar; in J. Jeuring et.al. (eds.), Intelligent Computer Mathematics, LNCS 7362, pp. 453-457, 2012.

[Kor13] A. Korniłowicz, On Rewriting Rules in Mizar; Journal of Automated Reasoning, vol. 50(2), pp. 203-210, 2013.

[Miz14] The Mizar Home Page; `http://mizar.org`.

[Poc14] H.C. Pocklington, The Determination of the Prime or Composite Nature of Large Numbers by Fermat's Theorem; Proc. Camb. Phil. Soc., vol. 18, pp. 29-30, 1914.

[Ric06] M. Riccardi, Pocklington's Theorem and Bertrand's Postulate; Journal of Formalized Mathematics, vol. 14(2), pp. 47-52, 2006.

[RST01] P. Rudnicki, A. Trybulec, C. Schwarzweller, Commutative Algebra in the Mizar System; Journal of Symbolic Computation, vol. 32(1/2), pp. 143-169, 2001.

[Sch07] C. Schwarzweller, Mizar Attributes: A Technique to Encode Mathematical Knowledge into Type Systems; Studies in Logic, Grammar and Rhetoric, vol. 10(23), pp. 387-400, 2007.