Designing Mathematical Libraries based on Requirements for Theorems

Christoph Schwarzweller
(`schwarzw@informatik.uni-tuebingen.de`)
Wilhelm-Schickard-Institute for Computer Science
University of Tübingen
Sand 13, D-72076 Tübingen, Germany

Abstract. Theorems can be considered independent of abstract domains; a theorem rather depends on a set of properties necessary to prove the theorem correct. Following this observation theorems can be formulated and proven more generally thereby improving reuse of mathematical theorems. We discuss how this view influences the design of mathematical libraries and illustrate our approach with examples written in the Mizar language. We also argue that this approach allows for checking whether particular instantiations of generic algorithms are semantically correct.

Keywords: Mechanized Reasoning, Reuse of Theorems, Requirements for Theorems, Mathematical Libraries, the Mizar System

AMS classification: 68W30, 68N30, 68T15, 68T30, 68T35

1. Introduction

At the end of the 19th century the interest of mathematicians turned away from concrete objects as e.g. numbers or points and lines. Since then more abstract domains like groups, rings, or vector spaces given by axioms only have been considered. The advantage is obvious and well-known: besides abstracting away from unnecessary details, this allows to prove theorems in an abstract setting. So once a theorem is proven for an abstract domain it holds in every particular domain fulfilling the axioms of the abstract one. This can be considered as formulating requirements for theorems: a special domain must fulfill the abstract domain's properties in order that the theorem can be applied. However, this is not the end of the line: often a theorem proven in an abstract domain does not depend on all properties given in the domain's definition; this can be observed in the corresponding proof where not all properties are actually used. So we claim that the proper basis of a theorem is a set of properties rather than an abstract domain. Consequently, theorems can be considered independent of domains: they exist in their own merit, and there are different abstract domains in which a theorem holds.

Using mechanized reasoning systems numerous theorems have been proven with machine assistance: mathematical facts, even such involved ones as e.g. the Jordan curve theorem [12] or Robbin's conjecture [7] have been verified or indeed been solved mechanically. However, in mechanized reasoning systems theorems are usually proven with respect to fixed abstract (or nonabstract) domains, too. This interferes with reuse of theorems and proofs and is, as already mentioned, an unnecessary restriction.

Libraries of proven theorems are not much developed yet. In fact some libraries exist, but most of them are not much more than a collection of proven theorems; they lack special features for processing theorems, in particular for reusing theorems in different domains. We believe that proving theorems on the one hand, and storing and processing theorems in a library on the other hand, are two distinct tasks: the main purpose of a theorem prover is to verify the correctness of theorems; in contrast the main purpose of a library is to collect and, more importantly, to allow reusing theorems in probably different domains. We aim at an approach supporting easy and semantically correct reuse of theorems. Easy means that reusing a theorem already proven should work without giving a (detailed) proof that or why a theorem holds in a different domain. Nevertheless theorems should only be reusable in a domain if this is semantically correct. We believe that these issues require a special representation of theorems and domains focussing on library demands. To this end we propose a properties based approach: a theorem should be formulated with respect to a set of properties necessary to prove it. The validity of a theorem in a particular domain then reduces to checking whether the theorem's properties hold in the domain. Hence, the theorem itself need not be dealt with again.

The Standard Template Library (STL) [9] has shown that it is possible to design libraries heavily supporting reusability. Parts of STL algorithms are left abstract using so-called template parameters [18], which are then instantiated with different types. In this way generic algorithms are constructed supporting code reuse. However, to result in an algorithm working correctly, an instantiation has to fulfill certain requirements: operations have to be present and, more importantly, semantic properties of these operations are assumed by generic algorithms. Unfortunately, semantic requirements are not checked when working with the STL, so that errors are detected at runtime only. To overcome these shortcomings is the goal of the SuchThat project [14, 8], in which the specification of generic algorithms became part of the programming language itself: a formal description of requirements so that the algorithm works is included. Checking whether an instantiation of a generic algorithm is correct is thus nothing else than checking whether a theorem holds in a particular domain. Hence we claim that the problem of correct instantiation of generic algorithms should also be handled using a properties based approach for theorems.

The plan of the paper is as follows. In the next section we present our properties based view on theorems in detail and discuss its impact on the structure of mathematical libraries. Demands for mathematical libraries and reasoning systems used are derived. In section 3 and 4 we show how the Mizar Language [12, 10] allows to systematically develop theorems and domains based on properties. We present some examples from (commutative) algebra, in particular from ideal theory. These examples also illustrate how reuse of theorems is supported due to the emphasis of properties. Section 5 is devoted to generic algorithms: we consider the Euclidean algorithm for computing greatest common divisors and prove its correctness following our approach. This shows that greatest common divisors can be computed using Euclid's method in domains where, in particular, neither multiplication nor addition is commutative. In the last section we discuss some problems of the subject and give hints for further work.

2. Requirements for Theorems

In most mechanized reasoning systems properties are solely used for the description of domains, not of theorems. Domains are defined by a set of axioms whereas theorems are stated and proven with respect to domains, hence with respect to a fixed set of axioms. However this is an unnecessary restriction, because the validity of a theorem is not bound to a particular domain, but depends on whether certain properties of the operators involved are true. Therefore we propose to consider properties as the basic ingredients of theorems. Focussing on properties allows to describe theorems using as few properties as possible: only carriers and operators necessary to formulate the theorem's content and properties necessary to prove the theorem's correctness are stated. Thus properties based theorems are more general, and moreover properties used in proofs can be considered as requirements domains have to meet so that a theorem is valid.

Let us consider a straightforward example: an ideal is conventionally defined over a (commutative) ring R; it is a subset S of R that is closed with respect to both addition and multiplication with arbitrary ring elements, that is for all $a, b \in S$ and all $r \in R$ we have $a + b \in S$ and both $r \cdot a \in S$ and $a \cdot r \in S$. It is rather obvious that the set $\{0\}$ is an ideal for an arbitrary ring R where 0 denotes the ring's zero element; so we have the following theorem.

> *Let R be a (commutative) ring. Then $\{0\}$ is an ideal in R.*

However, to prove the theorem by the definition of an ideal it is sufficient to show $0 + 0 = 0$ and $a \cdot 0 = 0 = 0 \cdot a$ for all $a \in R$. A closer inspection shows that this can be done only assuming that addition is associative, provides a right zero as well as a right complement and that addition and multiplication are distributive. This gives rise to the following theorem.[1]

> *Let $R = (R, +, *, 0)$ be a domain such that $+$ is associative, 0 is a right zero with respect to $+$ and $+$ is right complementary. Furthermore, assume that $+$ and $*$ distribute. Then $\{0\}$ is an ideal in R.*

Thus $\{0\}$ is an ideal in more general domains than rings. In particular to conclude that $\{0\}$ is an ideal in a given domain, as e.g. the integers, it is not necessary to prove that the domain is a ring: it suffices that the domain fulfills the four properties mentioned above.

In addition theorems based on properties are more robust concerning generalization of domains, just because unnecessary properties are not included. Consider again the theorem from above. Suppose it has been proven in its first version for commutative rings, and noncommutative rings are introduced. Then it has to be checked whether (the proof of) the theorem uses commutativity of multiplication. Hence the proof of the theorem itself has to be taken into consideration again. This is not necessary for the second, properties based version of the theorem because commutativity of multiplication is no property the theorem relies on; the theorem holds because noncommutative rings fulfill all properties required for (the proof of) the theorem.

Domains can also be considered as established by a set of their operations' properties. This holds for both abstract and nonabstract domains. Of course the definition of operations in nonabstract domains is crucial for the validity of properties. However, once these properties have been proven the operations' definitions become less important: whether a theorem holds can be expressed solely using properties. Consider for instance the integers. Whether the example theorem holds only depends on the properties of integer addition and multiplication, namely on whether integer addition is associative, right complementary and provides a right zero, and on whether integer addition and multiplication distribute. Thus, if a sufficient set of properties is already

[1] Provided that the notion of ideals has been introduced not only for (commutative) rings but for the more general domain also.

proven, the actual definitions have not to be taken into account again: the validity of a theorem in a particular domain can be reduced to checking whether the domain fulfills certain properties.

Summarizing we claim that theorem proving can be decomposed into two portions: proving general theorems based on properties on the one hand, and proving properties of particular domains on the other hand. Consequently, a mathematical library, and in particular the mechanized reasoning system used, should provide the possibility to

– define properties independently of domains

– combine properties in order to use them in proofs of theorems

– define specific domains by characterizing them in terms of properties

– infer validity of theorems in domains by comparing sets of properties.

Thereby it is rather a matter of convenience than of necessity whether a reasoning system supports the last item. The first three items, however, not only allow to formulate and prove theorems in a more general setting, but also to check the validity of a theorem in a particular domain by just comparing the theorem's and domain's properties.

3. Formalizing Requirements for Theorems in Mizar

In the following we illustrate the properties based approach using the Mizar system [10]. The Mizar Mathematical Language and its corresponding proof language were designed to reflect mathematical vernacular. Thus Mizar provides the features we discussed in the last section: properties of operators can be described independently of domains, that is with respect to a signature only. These properties then can be combined to prove theorems and define domains. Again we use the theory of ideals and the theorem from the last section as an example.[2]

First, we have to deal with the signature of a theorem, that is we have to fix carriers and operators used in the theorem's content. For that purpose, we use the language construct *structure definition*. Rings, for example, are given by a carrier, two binary and two nullary operators, the latter ones denoting the zero and the unit element. Hence, the typical signature for rings—called in Mizar `doubleLoopStr` (see [6])—can be defined by

[2] The basics of the theory of ideals have been formalized in Mizar and are part of the Mizar Mathematical Library; see [1].

```
definition
struct (LoopStr,multLoopStr_0) doubleLoopStr
   (# carrier     -> set,
      add, mult   -> BinOp of the carrier,
      unity, Zero -> Element of the carrier #);
end;
```

The terms LoopStr and multLoopStr_0 mentioned in the first line of the definition denote (already defined) Mizar structures. LoopStr provides a carrier with one binary operator add and a zero element Zero, multLoopStr_0 a carrier with one binary operator mult and a unit element unit. These two structures are now glued together yielding doubleLoopStr. As a consequence doubleLoopStr is in particular both a LoopStr and a multLoopStr_0 and all notions that were defined for these are also available for doubleLoopStr.

Now specific properties of one or more operators are introduced by defining *attributes* over structures providing the necessary signature. Note that in the following definition associativity of addition[3] is introduced for LoopStr only, just because a multiplication operator is not necessary here. Nevertheless, the attribute add-associative will be available for doubleLoopStr also, as LoopStr is an ancestor of doubleLoopStr.

```
definition
let R be non empty LoopStr;
attr R is add-associative
  for x,y,z being Element of R holds x+(y+z) = (x+y)+z;
end;
```

Further properties of structures can be introduced analogously, for example right_zeroed, right_complementable and distributive, which are further properties necessary to state the example theorem of section 2. Please note, that the attribute right_zeroed as well as the attribute right_complementable can be defined also for LoopStr, whereas distributive requires a doubleLoopStr, namely an addition and a multiplication operator.

In the same way properties necessary for a subset S of R to be an ideal over R can be introduced using attributes. Again, in the first definition R is a LoopStr, that is R provides no multiplication. Similarly, the other two properties are defined for HGrStr—an ancestor of multLoopStr_0— giving a carrier and multiplication only.

[3] The operator + is just a shorthand for the binary operator add of a structure; it uses hidden arguments, so that the carrier R of the operator + need not be explicitly stated.

```
definition
let R be non empty LoopStr, S be Subset of R;
attr S is add-closed means
  for x,y being Element of R st x in S & y in S
  holds x+y in S;
end;
```

```
definition
let R be non empty HGrStr, S be Subset of R;
attr S is left-ideal means
  for a,x being Element of R st x in S holds a*x in S;
attr S is right-ideal means
  for a,x being Element of R st x in S holds x*a in S;
end;
```

Combining these three properties we get the notion of an ideal.[4] Here we need the signature given by `doubleLoopStr`, as now both an addition and a multiplication operator must be provided. It may be worth mentioning that using our approach we have introduced the notion of an ideal for much weaker domains than rings; in fact we just used the signature of rings, but no further properties of the given operators were necessary. Of course, as we shall see, each theorem usually requires some ring properties in order to be proven.

```
definition
let R be non empty doubleLoopStr;
mode Ideal of R is add-closed
       left-ideal right-ideal (non empty Subset of R);
end;
```

We like to mention that the Mizar system expects an existence proof for such an attributed structure. Otherwise it cannot be used in theorems and further definitions. This is to avoid (theorems about) empty modes as for instance `empty infinite set`. In our example one has to show that for an arbitrary object `R` of type `non empty doubleLoopStr` there exists an object of type `non empty Subset of R` fulfilling the attributes `add-closed`, `left-ideal` and `right-ideal`.

Now we can formalize the theorem mentioned in section 2 by just combining the signature, i.e. the appropriate Mizar structure, with the required properties, i.e. the required attributes:

```
theorem T0:
for R being add-associative right_complementable
            right_zeroed distributive
            (non empty doubleLoopStr)
holds {0.R} is Ideal of R;
```

[4] Note that by combining the same properties differently other notions can be introduced, in our case e.g. left and right ideals.

Note that R provides exactly the properties necessary to prove the theorem.[5] So the theorem is valid not only in rings, but also in more general domains. In addition, as already mentioned, the theorem is easier applicable for a particular domain (even if the domain is in fact a ring): we have to verify only four properties to apply it, whereas it takes to prove eight properties to show that a domain is a ring. In general this phenomenon does not only hold for domains, but also for defined notions as the following example shows. Having the notion of ideals (in rings), the following definition of the quotient of ideals is straightforward: let R be a (commutative) ring, and let I and J be ideals of R. The quotient $I\%J$ of I and J is given by $\{a \in R \mid a \cdot J \subseteq I\}$, where $a \cdot J = \{a \cdot j \mid j \in J\}$. Now, if the sum $\{i+j \mid i \in I,\ j \in J\}$ of two ideals I and J is denoted by $I + J$, one can easily prove the following theorem:

> *Let R be a (commutative) ring, and let I,J and K be ideals of R. Then I % (J + K) = (I % J)* $\cap$ *(I % K) holds.*

Again, to prove the theorem it is not necessary that R is a commutative ring. But the proof also does not rely on I, J and K fulfilling all the properties of an ideal, that is being `add-closed`, `left-ideal` and `right-ideal`: it suffices that I is closed with respect to addition and that J and K are closed with respect to right multiplication with arbitrary elements. So, in [1] the following theorem has been proven:

```
theorem
for R being left_zeroed right_zeroed add-right-cancelable
            right-distributive (non empty doubleLoopStr),
    I being add-closed (non empty Subset of R),
    J, K being right-ideal (non empty Subset of R)
holds I % (J + K) = (I % J) /\ (I % K);
```

Note, that from the properties of R, I, J and K that have been used in this version of the theorem, it automatically follows that the theorem also holds for right ideals in noncommutative rings.

4. Reusing General Theorems in Mizar

In the last section we saw how properties based theorems can be formalized using the Mizar language. Now we explain how particular domains are constructed and how general theorems can be applied to such domains. As we will see in Mizar both abstract and nonabstract

[5] The proof of the example theorem has been carried out and can be found in [1].

domains are essentially a structure combined with a set of attributes. Consequently, the validity of general theorems in particular domains can be checked by comparing these sets of attributes.

Defining abstract domains is straightforward. An abstract ring, for example, can be defined the same way as ideals in section 3: we just combine the necessary structure (signature) with the necessary attributes (properties):

```
definition
mode Ring is Abelian add-associative right_zeroed
  right_complementable associative left_unital
  right_unital distributive (non empty doubleLoopStr);
end;
```

Note that this definition again requires an existence proof. The example theorem `T0` from section 3 is true for `Ring`s: the underlying structures of `Ring` and `T0` are the same and `T0`'s attributes are a subset of the attributes a `Ring` fulfills. And in fact the Mizar checker accepts the following

```
theorem
for R being Ring holds {0.R} is Ideal of R by T0;
```

by just referencing the general theorem `T0` from section 3. Please note again, that it was sufficient to define the notion of ideals for the structure `doubleLoopStr` in section 3. The simple fact that the definition of `Ring` is based on a `doubleLoopStr` (and that no attributes were used to define ideals) guarantees that the notion of ideals is also available for `Ring`s.

Constructing specific domains is a bit more elaborate because we have to take into account the definitions of operations. Let us consider the integers as an example.[6] First we have to provide the signature of the integers. Here we consider the integers as an algebraic domain, that is the signature is a `doubleLoopStr` giving a carrier two binary operations `add` and `mult` and two elements of the carrier, a `unity` and a `Zero`. Further components of the integers can be considered, e.g. to formalize an ordering, but for our purpose a `doubleLoopStr` is sufficient. Then we have to define the set of integers `INT` as well as addition and multiplication of the integers which are Mizar functors denoted by `addint` and `multint`. Then we can introduce the (ring of) integers `INT.Ring` as a Mizar functor yielding a `non empty doubleLoopStr`, in which the components are identified with the just mentioned carrier and operators.[7]

[6] Compare [15] where the ring of integers is defined using the Mizar system.
[7] The term `in INT` is a type coercion ensuring that `1` and `0` are indeed elements of the set `INT` as required by the structure definition.

```
definition
func INT.Ring -> non empty doubleLoopStr equals
  doubleLoopStr(#INT,addint,multint,1 in INT,0 in INT#);
end;
```

Note, that this definition only glues together the set `INT` with some operations on this set; no properties of the operators are included. In order to apply general theorems to the integers we now have to prove that certain attributes are fulfilled. In Mizar this is best done using *functorial cluster definitions.* Of course, the fact that e.g. the addition of `INT.Ring` is associative can also be stated (and proven) as a theorem. However, clustering has the advantage, that attributes are linked to the structure: using a cluster definition the Mizar checker can automatically infer that `INT.Ring` fulfills the clustered attribute. The number and the order of attributes in a cluster definition is without meaning and up to the user. In our example all attributes necessary for `INT.Ring` to be a (commutative) ring are put in one cluster definition:

```
definition
cluster INT.Ring ->
  Abelian add-associative right_zeroed
  right_complementable well-unital associative
  commutative distributive;
end;
```

It may be worth mentioning that Mizar expects a (coherence) proof for such a cluster definition: one has to prove that the domain indeed fulfills the attributes clustered. After the cluster has been registered the Mizar checker accepts the following integer version of the theorem from section 3 by just referencing `T0`. As in the case of `Ring`s this is due to the fact that the structures involved are both of type `doubleLoopStr` and the theorem's attributes are a subset of the attributes `INT.Ring` fulfills due to (the proofs of) the clusters.

```
theorem
{0.(INT.Ring)} is Ideal of INT.Ring by T0;
```

Note also, that the above cluster definition provides more properties of the integers than necessary for the validity of the theorem. For that, it is sufficient to show that `INT.Ring` is `add-associative right_zeroed right_complementable` and `distributive`.

We close this section by presenting another Mizar language construct allowing to express—and automate—implications of (sets of) attributes, *conditional cluster definitions.* For instance, if a domain is right-distributive and multiplication is commutative, then the domain is obviously left-distributive, too. The corresponding cluster definition looks as follows.

```
definition
cluster commutative right-distributive ->
                left-distributive (non empty doubleLoopStr);
end;
```

The key point is that a conditional cluster definition enriches the Mizar checker: notations or theorems, that have been defined or hold for structures fulfilling the attribute `left-distributive`, now are defined or hold for domains fulfilling the attributes `right-distributive` and `commutative` also. Thus conditional cluster definitions can reduce the number of attributes that has to be proven for a domain in order to check the validity of a theorem.

5. Requirements for Generic Algorithms

A generic algorithm can be considered as an algorithmic scheme: parts of the algorithm are left abstract using some kind of type parameter. These abstract parts may be concerned with data handling [9] or even with domains the algorithm is dealing with [14]. To get a running algorithm the abstract parts have to be instantiated with concrete pieces of code. However, it is usually not guaranteed that after instantiation the resulting nongeneric algorithm works correctly: the instantiation may lack necessary operations or the operations do not fulfill all requirements implicitly given by the generic algorithm. Consider for example a sorting algorithm; a generic version can be formulated for an arbitrary domain with a binary relation. However, to actually sort sequences over a particular domain it is necessary that the binary relation is a total order. In other words, if the instantiation for the generic sorting algorithm does not fulfill the requirement that its associated binary relation is a total order, the instance of the generic sorting algorithm will not work correctly.

Consequently, the correctness of a generic algorithm can be formulated with respect to a set of properties of the algorithm's operations: provided that a particular instantiation fulfills these properties, the resulting nongeneric algorithm will work correctly. Thus the problem of correct instantiation is nothing else than checking whether a properties based theorem is valid in a particular domain (which is implicitly given by the instantiation). In our example, properties necessary for the correctness of the generic sorting algorithm are reflexivity, antisymmetry, transitivity and totality of the binary relation. The integers enriched with their usual order provide these properties, hence the integers constitute a legal instantiation for a generic sorting algorithm.

In the following we present a case study on properties based verification of generic algorithms: we use the Mizar system to formalize and prove the correctness of Euclid's algorithm for computing greatest common divisors. It is well-known that Euclid's method works for arbitrary Euclidean domains [2]. However, our proofs will show that Euclidean domains provide more properties than necessary to make the method work.

We model the Euclidean algorithm by the sequence `e-seq` of remainders computed. However, at this point it is irrelevant how—or even whether—remainders are computed. Thus we can define `e-seq` as a function that, based on two initial values `a` and `b`, computes the next value by just applying a given function `g` to the two preceding values. The computation proceeds as long as the second argument of `g` does not equal `0.R`.

```
definition
let R be non empty ZeroStr,
    g be Function of [:R,R:],R,
    a,b be Element of R;
func e-seq(a,b,g) -> Function of NAT,R means
  it.0 = a & it.1 = b &
  for i being Nat holds
  it.(i+1) = 0.R or it.(i+2) = g.(it.(i),it.(i+1));
end;
```

Next we define two requirements describing the correctness of Euclid's method. First, the computation should terminate, in other words `e-seq` should eventually yield the value 0. Second, the greatest common divisor of the initial values `a` = `e-seq.0` and `b` = `e-seq.1` should be invariant throughout the computation, that is two consecutive pairs of values in `e-seq` should possess the same greatest common divisor. Note however, that the attributes are defined for arbitrary functions `f` and not for `e-seq` only.

```
definition
let R be non empty ZeroStr,
    f be Function of NAT,R;
attr f is terminating means
  ex t being Nat st t > 0 & f.t = 0.R;
end;

definition
let R be non empty doubleLoopStr,
    f be Function of NAT,R;
attr f is gcd_computing means
  for c being Element of R, i being Nat holds
  f.(i+1) = 0.R or
```

```
   (c is_gcd_of f.i,f.(i+1) implies
                      c is_gcd_of f.(i+1),f.(i+2));
 end;
```

Thus proving that `e-seq(a,b,g)` is `terminating` and `gcd_computing` for all initial values `a` and `b` shows the correctness of Euclid's method for arbitrary domains `R` with arbitrary degree function `d` and arbitrary function `g`. However, to do so further properties of the domain `R` (and of the function `g`) are necessary. Essential for termination is the existence of a degree function well-known from Euclidean rings. Here we define degree functions isolated from other ring properties by first introducing a corresponding attribute. We use a somewhat unusual form of the Euclidean property which we called `Left-Euclidean`: `a` is decomposed into `r + q * b`, which differs from `q * b + r` if addition is not commutative. This allows to prove the requirements from above without assuming commutativity of addition.[8] Note however, that in case addition is commutative our definition coincides with the usual one for Euclidean domains which can be expressed as a Mizar cluster definition.

```
definition
let R be non empty doubleLoopStr;
attr R is Left-Euclidean means
  ex f being Function of the carrier of R,NAT st
  for a,b being Element of R st b <> 0.R holds
  ex q,r being Element of R st
  a = r + q * b & (r = 0.R or f.r < f.b);
end;
```

```
definition
let R be Left-Euclidean (non empty doubleLoopStr);
mode DegreeFunction of R
         -> Function of the carrier of R,NAT means
  for a,b being Element of R st b <> 0.R holds
  ex q,r being Element of R st
  a = r + q * b & (r = 0.R or it.r < it.b);
end;
```

Finally, we need to formalize that the function `g` used by `e-seq` computes remainders. This can be easily done for arbitrary degree functions of `R`, hence for arbitrary structures that are `Left-Euclidean` as the following definition shows. Note however, that it is possible to require other properties for `g`. Crucial is that the properties required allow to show that the resulting `e-seq` is `gcd_computing`.

[8] We generalized the well-known correctness proof found in text books (see for example [2]). This proof requires commutative addition if `a=q*b+r` instead of `a=r+q*b` is used. However, there may exist a different proof showing the correctness of Euclid's method without assuming commutativity of addition in this case.

```
definition
let R be Left-Euclidean (non empty doubleLoopStr),
    d be DegreeFunction of R,
    g be Function of [:R,R:],R;
pred g computes_mod_wrt d means
  for a,b being Element of R st b <> 0.R holds
  ex q being Element of R st a = g.(a,b) + q * b &
                   (g.(a,b) = 0.R or d.(g.(a,b)) < d.b);
end;
```

After these preparations we can prove the following theorems describing the correctness of Euclid's method for computing greatest common divisors. Not surprising the property `Left-Euclidean` is sufficient to prove termination of `e-seq` (provided of course that the function used to compute the sequence fulfills the property `computes_mod_wrt`). Theorem `T2` is more interesting: the properties used to prove the theorem show that greatest common divisors can be calculated[9] in domains `R` where neither addition nor multiplication is commutative. Furthermore `R` need not be an integral domain, that is `R` may contain zero divisors. Note also, that the (proofs of the) theorems holds for arbitrary degree functions `d` of `R`.

```
theorem T1:
for R being Left-Euclidean (non empty doubleLoopStr),
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R
holds e-seq(a,b,g) is terminating;

theorem T2:
for R being add-associative associative right-zeroed
            right_complementable left-distributive
            Left-Euclidean (non empty doubleLoopStr),
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R
holds e-seq(a,b,g) is gcd_computing;
```

Similar to section 4 we can now use theorems `T1` and `T2` to infer (generic and nongeneric) domains with which the instantiation of the generic Euclidean algorithm is legal. Assuming, for example, that the type `EuclideanRing` has been introduced in Mizar as a `doubleLoopStr` and that the appropriate attributes have been clustered, we get the following theorem.

[9] To actually compute greatest common divisors it is also necessary that `a` is a greatest common divisor of `a` and `0.R`. To prove this, addition of `R` must be left cancelable and `1.R` must be a left unity with respect to multiplication.

```
theorem
for R being EuclideanRing,
    d being DegreeFunction of R,
    g being Function of [:R,R:],R st g computes_mod_wrt d
for a,b being Element of R
holds e-seq(a,b,g) is gcd_computing &
      e-seq(a,b,g) is terminating by T1,T2;
```

Please note again that this theorem is accepted by just referencing theorems `T1` and `T2`, because the attributes of an `EuclideanRing` are a superset of the ones used in the theorems.

Consider the integers as a second example. The ring of integers in particular establishes a Euclidean domain. This is again formalized in a conditional cluster definition where it is shown that `INT.Ring` fulfills the attributes not proven so far.[10] After the cluster is registered the theorem that Euclid's algorithm instantiated with the integers is correct, can be proven by just referencing `T1` and `T2`.

```
definition
cluster INT.Ring -> Euclidean;
end;

theorem
for d being DegreeFunction of INT.Ring,
    g being Function of [:INT.Ring,INT.Ring:],INT.Ring
      st g computes_mod_wrt d
for a,b being Element of INT.Ring
holds e-seq(a,b,g) is gcd_computing &
      e-seq(a,b,g) is terminating by T1,T2;
```

The same way further parameters occurring in the theorem can be specialized easily. For instance, the usual absolute value function `absint` is a degree function of the ring of integers. Now, if `absint` is defined as a `DegreeFunction of INT.Ring`—which in Mizar includes a proof that `absint` indeed is a degree function—the following is accepted by the Mizar checker.

```
theorem
for g being Function of [:INT.Ring,INT.Ring:],INT.Ring
      st g computes_mod_wrt absint
for a,b being Element of INT.Ring
holds e-seq(a,b,g) is gcd_computing &
      e-seq(a,b,g) is terminating by T1,T2;
```

[10] Note that in the cluster the usual property `Euclidean` and not `Left-Euclidean` is proven. This is sufficient due to a conditional cluster definition stating that for commutative addition `Euclidean` implies `Left-Euclidean`. The same holds for the attributes `distributive` and `left-distributive`.

Thus a properties based approach allows to state the correctness of generic algorithms at a very abstract level, namely by formalizing requirements so that the algorithm works. Once the correctness or other properties of a generic algorithm have been shown with respect to these requirements, it is then possible to identify correct instantiations by just comparing properties of theorems and domains.

6. Conclusion and Future Work

We have described a properties based approach for stating and proving theorems and discussed its impact on designing libraries. Using properties instead of domains as the basis for theorems allows to prove more general theorems. The properties used in proofs can be considered as requirements for a theorem to be valid. Consequently, once a theorem has been proven with respect to a set of requirements, checking whether the theorem is valid in a particular domain reduces to checking whether the domain fulfills these requirements. We have illustrated our approach with examples from commutative algebra and generic programming using the Mizar system, a system that meets the demands necessary to handle and reuse properties based theorems.

As already mentioned, most mechanized reasoning systems deal with one theory at a time only. Definitions and theorems are developed with respect to a fixed set of axioms. The interplay of theorems (and definitions) in different theories is not addressed. This has been observed in [4] where so-called little theories have been introduced. The little theory approach proposes to work in different theories due to the amount of structure required. Theorems are then moved from one theory to another by means of theory interpretations [16], in this way allowing to reuse theorems. These ideas have been implemented in the IMPS system [5].

Another system supporting reuse of theorems by assuming properties necessary for proofs is Theorema [3]. In Theorema functors with abstract domains as parameters are used to construct new domains. Then theorems about functors, that is about the domain constructed, can be proven requiring certain properties of the parameterized domains. Thus these theorems hold for all domains matching the requirements used in the proof. Theorema is implemented on top of Mathematica [19].

However, using properties based theorems causes new questions: which kind of properties should be used to describe requirements for theorems? Recalling the example theorem from section 2, it is of course possible to define a property `0-ideal` combining all the requirements necessary to prove the theorem. Thus, it is always possible to find exactly one property describing the requirements of a theorem. That is not what we have in mind. We believe that the answer to this problem is given by the application domain: common notions used there, like e.g. right-associative and commutative in abstract algebra, correspond to the properties that should be used [17].

Another problem is that a set of minimal requirements for a theorem cannot always be found: the same theorem can be proven in more than one way by assuming different incomparable sets of properties. We illustrate this phenomenon with an example given in [13]: multiplication of elements of an algebraic domain with natural numbers can be defined in two ways, multiplication from the left and from the right. Then $n \cdot a$ stands for $a+(a+(\ldots(a+a)\ldots)$ and $a \cdot n$ for $(\ldots(a+a)+a)\ldots)+a)+a$. However in nonassociative domains $n \cdot a$ and $a \cdot n$ are not necessarily the same. Consequently, one starts with two definitions of this kind of multiplication, one for left- and one for right-multiplication. Then it is straightforward to prove that $n \cdot a$ equals $a \cdot n$ using that addition of the underlying domain is associative and that there exists a unity with respect to addition. However, another proof shows that this also holds if addition is just commutative. Now, a question comes up: what to do if the property $n \cdot a = a \cdot n$ is to be used in another proof? Obviously, there is the possibility of choosing an associative or a nonassociative domain, leading to different proofs, and hence again to two theorems with different requirements. It is hardly possible to predict which theorem will be more important for future work, so both seem to deserve their place in the library. On the other hand, storing two or more versions of the same theorem will ultimately bloat the library.

On the other hand using properties based theorems can result not only in generalization of domains but also of methods. For instance, the properties of the domain `R` used to show the correctness of Euclid's algorithm in section 5 were in fact necessary only to prove that $\gcd(a, b) = \gcd(b, \mathtt{g}(a, b))$ where `g` is the function yielding the remainder of a and b. The rest of our proofs is not affected by properties of `R`. Consequently, greatest common divisors can be computed in a domain `R`, if there are a function `g` with $\gcd(a, b) = \gcd(b, \mathtt{g}(a, b))$ and a well-founded relation $<$ with $\mathtt{g}(a, b) < b$ for all $a, b \in \mathtt{R}$. Properties of `R` allowing to prove this property of `g` are then sufficient to compute greatest common divisors in `R` independent of the function `g` realizes.

Acknowledgements: I would like to thank Rüdiger Loos, Andrzej Trybulec, Holger Gast, and Roland Weiss for many valuable discussions on the topic. The comments and suggestions provided by the referees have greatly improved the paper.

References

1. J. Backer, P. Rudnicki, and C. Schwarzweller, Ring Ideals, Formalized Mathematics 9 (2000) 565-583; available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol12/ideal_1.html
2. T. Becker and V. Weispfenning, Gröbner Bases—A Computational Approach to Commutative Algebra, Springer, 1993
3. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru, A Survey on the Theorema Project, in: Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation), ed. W. Küchlin, ACM Press, 1997, pp.384-391
4. W. M. Farmer, J. D. Guttman, and F. J. Thayer, Little Theories, in: Automated Deduction–CADE-11, ed. D. Kapur, LNCS 607, Springer, 1992, pp. 567-581
5. W. M. Farmer, J. D. Guttman, and F. J. Thayer, IMPS: An Interactive Mathematical Proof System, Journal of Automated Reasoning, 11 (1993) 213-248
6. E. Kusak, W. Leonczuk, and M. Muzalewski, Abelian Groups, Fields and Vector Spaces, Formalized Mathematics 1 (1990) 335-342; available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol1/vectsp_1.html
7. W. McCune, Solution of the Robbins Problem, Journal of Automated Reasoning 19 (1997) 263-276
8. D. Musser, The Tecton Concept Description Language (1998); available by anonymous ftp from http://www.cs.rpi.edu/~musser/gp/tecton
9. D. Musser, G. Derge, and A. Saini, STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library, Addison-Wesley, 2001
10. P. Rudnicki and A. Trybulec, On Equivalents of Well-foundedness. An Experiment in Mizar, Journal of Automated Reasoning 23 (1999) 197-234
11. P. Rudnicki and A. Trybulec, Multivariate Polynomials with Arbitrary Number of Variables, Formalized Mathematics (1999), to appear; available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol11/polynom1.html
12. P. Rudnicki and A. Trybulec, Mathematical Knowledge Management in Mizar, in: Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM2001), Linz, Austria, 2001; available by anonymous ftp from http://www.risc.uni-linz.ac.at/institute/conferences/MKM2001/Proceedings/
13. P. Rudnicki, C. Schwarzweller, and A. Trybulec, Commutative Algebra in the Mizar System, Journal of Symbolic Computation 32 (2001) 143-169
14. S. Schupp and R. Loos, SuchThat—Generic Programming Works, in: Generic Programming—International Seminar on Generic Programming, eds. M. Jazayeri, R. Loos and D. Musser, LNCS 1766, Springer, 1998, pp. 133-145
15. C. Schwarzweller, The Ring of Integers, Euclidean Rings and Modulo Integers, Formalized Mathematics 8 (1999) 17-22; available by anonymous ftp from http://mizar.uwb.edu.pl/JFM/Vol11/int_3.html
16. J. R. Shoenfield, Mathematical Logic, Addison-Wesley, 1967
17. I. Sommerville, Software Engineering, 4th edition, Addison-Wesley, 1992
18. B. Stroustrup, The C++ Programming Language, 3rd edition, Addison-Wesley, 1997
19. S. Wolfram, The Mathematica Book, 4th edition, Addison-Wesley, 1999