Scratchpad's View of Algebra I: Basic Commutative Algebra

J.H. Davenport
School of Mathematical Sciences
University of Bath BA2 7AY
Claverton Down
Bath
England

and

B.M. Trager
Department of Mathematical Sciences
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights
10598 NY, U.S.A.

Abstract. While computer algebra systems have dealt with polynomials and rational functions with integer coefficients for many years, dealing with more general constructs from commutative algebra is a more recent problem. In this paper we explain how one system solves this problem, what types and operators it is necessary to introduce and, in short, how one can construct a computational theory of commutative algebra. Of necessity, such a theory is rather different from the conventional, non-constructive, theory. It is also somewhat different from the theories of Seidenberg [1974] and his school, who are not particularly concerned with practical questions of efficiency.

Introduction

This paper describes the constructive theory of commutative algebra which underlies that part of Scratchpad which deals with commutative algebra. We begin by explaining the background that led the Scratchpad group to construct such a general theory. We contrast the general theory in Scratchpad with Reduce–3's theory of domains, which is in many ways more limited, but is the closest approach to an implemented general theory to be found outside Scratchpad. This leads us to describe the general Scratchpad view of data types and categories, and the possibilities it offers. We then digress a little to ask what criteria should be adopted in choosing what types to define. Having discussed the philosophical issues, we then discuss commutative algebra proper, breaking this up into the sections "up to Ring", "Integral Domain", "Gcd Domain" and "Euclidean Domain". It should be noted that, while most of the decisions taken in Scratchpad have a sound mathematical foundation, some, such as the decision not to define various devleopments of semi-groups, such as quasi-groups, are not so soundly based, and reflect the authors' prejudices as much as anything else. We will endeavour to distinguish these decisions from those with a more mathematical foundation. This brings us to the heart of the matter: the definitions used in Scratchpad. Since we are more interested in the theory than in the practical details of Scratchpad, we will often simplify the details of implementation. The full details can be found in Davenport & Trager [1990].

Do we need a new theory? Why can't Scratchpad just use the conventional definitions of, say, "ring", "field", "integral domain" or "unique factorisation domain" found in any text-book on abstract algebra? The reason is that these definitions are fundamentally non-constructive. They say that things exist, but do not give any algorithms for constructing them. Furthermore, such algorithms may well not exist. For example, it is well-known in abstract algebra that, in the presence of noetherianity, the existence of greatest common divisors is equivalent to the existence of unique factorisation. However, as was first shown by Fröhlich & Shepherdson [1956], there exist domains with algorithms for computing greatest common divisors, but for which there cannot exist algorithms for computing unique factorisation. Their example made use of a recursively enumerable, non-recursive sequence to generate a field K which might be $\mathbf{Q}$ or $\mathbf{Q}[i]$, but such that one couldn't tell which. $K[x]$ is certainly noetherian, since K is a field. Then it is certainly possible to compute greatest common divisors in $K[x]$, since Euclid's algorithm is purely rational

in its inputs. But, computing the factorisation of $x^2 + 1$ is equivalent to deciding what K is, and so is impossible. We will make use of several such constructions as we show why we need to make certain distinctions which the non-constructive theory doesn't make.

The Problem

The handling of polynomials with integer coefficients is one of the oldest problems of computer algebra [Collins, 1966]. The difficulties encountered in the implementation of polynomials with integer coefficients were largely ones of efficiency, especially for the computation of greatest common divisors and the factorisation of polynomials. While improvements continue to be made in these areas, it is fair to say that these problems are largely solved in principle, though the algorithms are not as easy to implement as one would like:

> *We found that, although the Hensel construction is basically neat and simple in theory, the fully optimised version we finally used was as nasty a piece of code to write and debug as any we have come across* [Moore & Norman, 1981].

Once $\mathbf{Z}[x_1, \ldots, x_n]$ has been implemented, it is possible to implement $\mathbf{Q}(x_1, \ldots, x_n)$ as the quotient field of $\mathbf{Z}[x_1, \ldots, x_n]$. To ensure canonical forms, we need merely verify that there is no common factor between the numerator and the denominator (hence one major use of the computation of gcds) and that the leading coefficient of the denominator is positive. $\mathbf{Q}[x_1, \ldots, x_n]$ is generally treated as a special case of $\mathbf{Q}(x_1, \ldots, x_n)$, in other words, a global denominator is used rather than local denominators. This is generally justified by arguing that the cost of repeated gcd calculations between the numerators and denominators of the rational numbers greatly outweighs the cost of carrying the lcm of the individual denominators as a global denominator. Hence, in principle, all questions of algebra with rational coefficients are solved.

In practice, things are not so simple. Let us consider Hermite's algorithm [Hermite, 1872] for the integration of a rational function $f(x)$ in $\mathbf{Q}(x)$ (or, more precisely, for finding the rational part of the integral of such a function). The algorithm (given fully in Davenport & Trager [1990]) is fundamentally based on the algebra of $\mathbf{Q}[x]$: the divisions, the partial fraction decomposition, the solution of Bézout's equality and the remainder calculations all take place in $\mathbf{Q}[x]$ rather that in $\mathbf{Z}[x]$. This is not to say that the algorithm cannot be implemented over $\mathbf{Z}[x]$: many authors have done that in many algebra systems. But the implementation becomes much more complicated: every variable must be replaced by a pair consisting of a polynomial in $\mathbf{Z}[x]$ and a denominator in $\mathbf{Z}$, and a twelve-line algorithm becomes several pages of code.

Worse, consider what happens when we have a parametric integral, so that $\mathbf{Q}[x]$ is replaced by $\mathbf{Q}(y)[x]$. We have to embed $\mathbf{Q}(y)[x]$ in $\mathbf{Q}(y, x)$, the quotient field of $\mathbf{Z}[y, x]$. If our system insists on treating y as the main variable, rather than x, in some recursive polynomial representation, or adopts a distributed representation, operations such as the initial synthetic division, which are mathematically trivial in $K[x]$ for any field K, become software nightmares, since the data structure is not representing the underlying mathematics.

More generally, there are many rings between $\mathbf{Z}[x_1, \ldots, x_n]$ and $\mathbf{Q}(x_1, \ldots, x_n)$, and many algorithms which are naturally set in one of these intermediate rings rather than in either of the extremities. Furthermore, there are quotients of these intermediate fields, such as algebraic number fields. Davenport [1981] describes the difficulty of trying to manipulate algebraic extensions in a system (Reduce-2) which was essentially purely polynomial.

Before we proceed much further in this direction, we should sound a note of warning. It is certainly true that the ability to talk about objects like $\mathbf{Q}(x_1, \ldots, x_n)[y]$ is useful when it comes to expressing algorithms. However, these algorithms may not be the most efficient possible. For example, it is possible to use Euclid's algorithm to compute greatest common divisors in $\mathbf{Q}(x_1)(x_2)\ldots(x_n)[y]$, regarding each extension (x_i) as the field of fractions of the polynomial extension $[x_i]$, and using Euclid's algorithm to calculate greatest common divisors every time fractions have to be added or multiplied. But it is far more efficient to clear denominators, and to use a modular or p-adic algorithm in $\mathbf{Z}[x_1, \ldots, x_n][y]$.

System Requirements

The two major computer algebra systems in which this has been treated are Scratchpad [Jenks & Trager, 1985] and Reduce–3 with its theory of domains [Bradford *et al.*, 1986]. These systems differ substantially in their approach to the problem: Scratchpad is a system designed *ab initio* to handle this view as abstractly as possible, whereas the theory of domains in Reduce–3 (more precisely Reduce–3.3) was intended as an extension to an existing successful system to enable it to handle a wider range of ground objects: earlier versions of Reduce were limited to polynomials (and rational functions) with integer or (machine-precision) floating-point coefficients, and the theory of domains extends this to allow user-defined types, as well as system-defined types such as "arbitrary-precision floating-point numbers", rational numbers, Gaussian integers (or Gaussian rationals, or Gaussian floating-point numbers or ...) and algebraic numbers. In the Reduce model, domains are either rings or fields, the sole difference being that division is always possible in fields, but not in rings. The whole of the "polynomial arithmetic" part (and packages which are based on it, such as the matrix manipulation package) of Reduce works with respect to any domain (except that the g.c.d. algorithm, which is sub-resultant based [Hearn, 1979] has severe problems with inexact domains, such as floating-point numbers), but packages such as factorisation and integration work with only a subset of the domains, with special-case code for each domain — for example, factorisation works directly with integers as the domain, converts rationals to integers first, and reduces Gaussian problems to non-Gaussian ones by taking the norm [Trager, 1976].

Scratchpad, on the other hand, allows any set of operators (and corresponding axioms) to form a definite class of types (the Scratchpad phrase is *category*), of which there are over 100 named ones currently defined in Scratchpad. Categories can in fact be parametrised by other types — the first instance of this in this paper is the definition of `LeftModule`, where the concept is explained. These are viewed as forming a multiple-inheritance hierarchy: a new category is defined as being the union of the operators and axioms of certain previously-defined categories, together with some new operators and axioms (of course, any of these components may be empty). We say that this category is the *direct descendant* of these previously-defined categories, which are the *direct ancestors* of the category just defined. The concepts *descendant* and *ancestor* are the reflexive-transitive closure of *direct descendant* and *direct ancestor* respectively.

New types (or domains: the two words are used almost interchangably in Scratchpad, but we will use "type" to avoid confusion with Reduce's theory of domains) are constructed by means of *functors*: functions which take some (possibly none) parameters, which may themselves be types, and return a new type. The parameters of a functor are themselves typed, so that an object is defined to come from some type, and a type from some category. For example, the type $\mathbf{Z}$ is created by applying the functor `Integer` (a function with no parameters), the type $\mathbf{Q}$ is created by applying the functor `Fraction` to the type $\mathbf{Z}$ (belonging to the category `IntegralDomain`), and the type* $\mathbf{Z}[x]$ is created by applying the functor `UnivariatePolynomial` to two arguments: the object x (belonging to the type `Symbol`) and the type $\mathbf{Z}$ (belonging to the category `Ring`).

A functor defines the implementation of the various operators that are defined on the resulting type. In general, the resulting type is defined to belong to a nonce category, generally a named one with some additional operations. For example, `Integer` could be defined to return an object that belonged to the category `EuclideanDomain` with an additional operator `positivep` that said whether or not the integer was positive (the actual definition is far more complicated). Hence the body of `Integer` would have to define operations such as `+`, `/` and `gcd` as well as `positivep`.

With such a rich language available, how do we decide which categories to define, and what functors should be available, and what categories should their arguments belong to? It is this question that this paper addresses, for that part of Scratchpad which implements commutative algebra. First, we must ask ourselves what criteria we should use to choose among the various

---

* More precisely, one of the many types in Scratchpad which is abstractly isomorphic to the abstract mathematical type $\mathbf{Z}[x]$. Other types can be created by using `DenseUnivariatePolynomial`, or by creating multivariate polynomials in only one variable, or in many other ways.

possibilities.

A Little Philosophy

Why does "abstract algebra" insist so much on the definition and use of concepts (*algebras* in the sense of the subject Universal Algebra: *categories* would be another word, and is the word Scratchpad has borrowed) such as "ring", "integral domain" and "field"? One answer, it seems to us, is *economy of effort*: for example, rather than proving many different theorems, such as "polynomials in one variable over the integers have a unique factorisation property", "polynomials in two variables over the integers have a unique factorisation property", "polynomials in two variables over the integers modulo 7 have a unique factorisation property" and so on, we need only prove one theorem — "polynomials in one variable over a unique factorisation domain form a unique factorisation domain". We will ask later whether this particular piece of generality can in fact be achieved constructively.

There are other reasons as well, which explain why a particular category is "successful". The first reason is one of *interest*: there must be some significant interest in various objects which belong to this category. Furthermore, the interest must have something to do with the property: for example, $\mathbf{Z}$ is interesting, as is $\mathbf{Z}/n\mathbf{Z}$ for odd n, but one is unlikely to find much interest in a theory of "rings which, when viewed as abelian groups, have an involution with precisely one fixed point".

Another reason is what we will call *functoriality*: there should be operations (functors) which construct new objects of the category from old objects of this category, or maybe from old objects of another category. For example, the functor $[x]$ (construct polynomials in one indeterminate over) takes integral domains into integral domains, and takes fields into Euclidean domains.

How does this translate into the computational setting? We certainly want *economy of effort*, by which we mean now that one implementation of an algorithm will work over several different types: for example one sorting algorithm working over all types belonging to the category `OrderedSet`. This is provided to some extent by the Reduce model, since the whole of polynomial arithmetic is provided over all domains by one piece of code (with the occasional dependence on whether the domain in question at the moment is a field or not). This would be easy to provide in Scratchpad, if all that was wanted were polynomial and rational function calculations over constant domains. But, as was pointed out in the introduction, we would like to see polynomials and rational functions defined over other domains, in particular over domains which are themselves polynomial or rational function domains.

We also want *interest*: it should be possible to implement difficult algorithms over many different types. For example, we would like to implement polynomial factorisation as few times as possible, and then have it operate over as wide a range of different types as possible. Hence we need to define a category such that:

a) It is possible to implement polynomial factorisation over this category;
b) As many types as possible belong to this category.

Such a goal may not be easy, but it is surely worth aiming for.

The types up to `Ring`

Scratchpad implements a fundamental category `SetCategory`, of which almost all other categories are descendants. Two operations are defined on types `$` (the standard Scratchpad notation for the type one is defining at the moment) belonging to this category

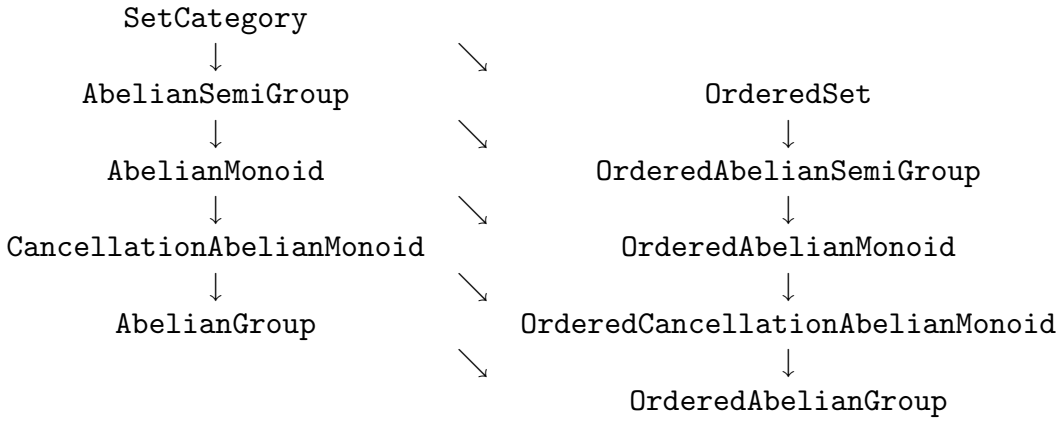
$$= : \$ \times \$ \mapsto \texttt{Boolean}$$

$$\texttt{coerce} : \$ \mapsto \texttt{OutputForm}$$

where `Boolean` is a built-in type of truth values, and `OutputForm` is a built-in type which is used in printing and other general-purpose expression-manipulation tasks. The assumption that almost all types contain an equality operator is extremely convenient for most purposes, though it could be argued that it is too restrictive. Note that we do *not* require that mathematical equality be

represented by Lisp equality, though it will generally be more efficient if this is the case. Domains in which Lisp equality is the same as mathematical equality are said to be *canonically represented*, and are declared to have the attribute `canonical`. This attribute is useful when it comes to considering the use of hashing, to quote but one example, since the hashing functions built into a Lisp system will not give the correct results unless the domain is canonically represented. Another way of viewing this attribute is to say that it asserts that objects that print (in terms of the coercion to `OutputForm`) differently really are different. We discuss the propagation of this attribute further in the section "What does it mean to be an Integral Domain?".

From this we can develop a straight-forward sequence which covers elementary commutative algebra:



where the arrows indicate a "direct descendant" relationship.

`AbelianSemiGroup` is defined to have one new operator:

$$+ : \$ \times \$ \mapsto \$$$

satisfying the associative and commutative axioms:

$$a + (b + c) = (a + b) + c$$
$$a + b = b + a.$$

`AbelianMonoid` introduces a new nullary operator

$$0 :\mapsto \$$$

satisfying the obvious axiom

$$0 + a = a.$$

`CancellationAbelianMonoid` is the category of abelian monoids with the cancellation axiom:

$$a + b = a + c \Rightarrow b = c.$$

Constructively, this is represented by a partial subtraction operator, whose signature is defined as:

$$- : \$ \times \$ \mapsto \texttt{Union}(\$, \text{"}\texttt{failed}\text{"}).$$

The right-hand side of $\mapsto$ is Scratchpad's notation for what other languages sometimes call a "disjoint union". `"failed"` is a distinguished symbol, which can be tested for by seeing which branch of the union is returned. While such an operation could be defined for any `AbelianMonoid`, or even any `AbelianSemiGroup` (as was indeed done in some earlier versions of Scratchpad), it is the cancellation axiom that ensures that $-$ has a unique value. This operator is subsumed in the $-$ operation defined on `AbelianGroup`s, so is not of immediate interest in the development of commutative algebra. When we come to define polynomial data types, we will rely on the existence of this operation in the exponent domain.

`AbelianGroup` adds one further unary operator:

$$- : \$ \mapsto \$.$$

This operator satisfies the axiom

$$a + (-a) = 0.$$

The first $\searrow$ introduces an operator

$$< : \$ \times \$ \mapsto \texttt{Boolean}$$

satisfying the usual axioms:

$$a < b \wedge b < c \Rightarrow a < c$$
$$\neg(a < b) \wedge \neg(b < a) \Rightarrow a = b$$
$$a < b \Rightarrow \neg(b < a).$$

Subsequent $\searrow$ in this diagram introduce no new operators, but one more axiom is introduced, when `OrderedAbelianSemiGroup` is defined:

$$a < b \Rightarrow a + c < b + c.$$

This is typical of what happens when two categories are merged to form a new named category: we keep the same operators, but are interested in the interaction between them, which requires the introduction of new axioms to define this interaction. Subsequent $\searrow$ in the chain represent the straight-forward merging of ancestors.

In Scratchpad, we have also defined types `SemiGroup` and `Monoid`, with the obvious multiplicative operations. We can now start defining ring-like objects properly. There is substantial disagreement (at the notational level) amongst mathematicians as to whether a ring need or need not contain a unity: we have chosen to require that a `Ring` needs to. Hence our first definition is of a `Rng`, which is defined to be both an `AbelianGroup` and a `SemiGroup`, with two additional axioms:

$$a * (b + c) = a * b + a * c$$
$$(b + c) * a = b * a + c * a.$$

If this domain has the property that the product of two non-zero elements is always non-zero, then we assert the additional attribute `noZeroDivisors`.

It would be pleasant to proceed now to the definition of a `Ring`, but we are caught here by a conflict between the Scratchpad requirement that a category be defined in terms of previously-defined categories, and the mathematical statement that *a ring is a (left-)module over itself.* We break the dilemma by defining a `NaiveRing` to be both a `Rng` and a `Monoid`, with operations

$$\texttt{characteristic} : \mapsto \texttt{NonNegativeInteger}$$
$$\texttt{recip} : \$ \mapsto \texttt{Union}(\$, \text{"}\texttt{failed}\text{"}).$$

The right-hand side of the last $\mapsto$ again includes a "disjoint union" and the distinguished symbol `"failed"`.

For an arbitrary `NaiveRing`, `characteristic` is defined as being the least positive integer n, if one exists, such that 1 added to itself n times is 0, otherwise 0. `recip` satisfies the axiom

$$\texttt{recip}(x) \neq \texttt{"failed"} \Rightarrow x * \texttt{recip}(x) = \texttt{recip}(x) * x = 1.$$

Clearly `recip` cannot be defined any earlier than this, since we need to have a definition of 1. It could be argued that the definition should be later, but it seems in practice to be convenient to define it here.

If R is any `NaiveRing`, we can define the category `LeftModule`(R) of all left-R-modules* to be sets `$` which are members of the category `AbelianGroup` equipped with an extra operation

$$* : R \times \$ \mapsto \$$$

and the corresponding axioms:

$$(a * b) * x = a * (b * x)$$
$$(a + b) * x = a * x + b * x$$
$$a * (x + y) = a * x + a * y.$$

* We could equally well have chosen to work in terms of right-R-modules.

The category `Ring` is then both a `NaiveRing` and a `LeftModule` over itself. A `Module` over a `Ring` R is then both a `LeftModule` and a `RightModule`.

A `CommutativeRing` is both a `Ring` and a `BiModule` over itself, with the additional axiom that multiplication is commutative.

What does it mean to be an Integral Domain?

The usual definition of an Integral Domain is rather non-constructive:

$$\nexists a, b \neq 0 : ab = 0.$$

Another way of saying this is to regard it as a property of multiplication: $a, b \neq 0 \Rightarrow ab \neq 0$. A third way is to see that it is much the same as "cancellation" in the type `CancellationAbelianMonoid`, since if $pq = pr$, then $p(q - r) = 0$, and if $p \neq 0$, then $q = r$. Knowing this property may well help in implementing an operation: for example the definition of multiplication of a sparse polynomial by an element of the underlying ring is defined as

```
if R has noZeroDivisors then
  r * x  ==
      r = 0 => 0
      r = 1 => x
      [[u.k,r*u.c] for u in x]
 else
  r * x  ==
      r = 0 => 0
      r = 1 => x
      [[u.k,a] for u in x | (a:=r*u.c) ^= 0$R]
```

where the knowledge of the `noZeroDivisors` property obviates the test to see whether any product has become zero. A `Ring` with this property is an `EntireRing`.

However, this is far from realising the full power of integral domains. For example, we would like to be able to implement Bareiss' [1968] fraction-free matrix algorithms, which are only valid over integral domain, not over general rings, and we would like to be able to implement quotient fields of integral domains. None of the definitions given above is very helpful from this point of view, though we can be inspired by the algorithmic rendering we gave "cancellation". We choose to give "integral domain" an algorithmic flavour by using the following corollary to the usual definitions: if R is an integral domain, then a/b, if it exists at all, is unique. Hence we choose to define an `IntegralDomain` to be a `CommutativeRing`, an `EntireRing` and an `Algebra` over itself, with an (infix) operator `exquo`:

$$\texttt{exquo} : \$ \times \$ \mapsto \texttt{Union}(\$, \text{"}\texttt{failed}\text{"}).$$

In this context, $a \,\texttt{exquo}\, b =$ `"failed"` should be interpreted as meaning "there is no element c of the current domain such that $bc = a$, but there's no reason why one shouldn't enlarge the domain to add one". For many domains, in particular euclidean domains, `exquo` could be defined in terms of a "quotient and remainder" operation, but it is often not very efficient to calculate an enormous remainder and then discover that it is non-zero*. `exquo` gives a hard error if the second argument is zero, since then there is no legal enlargement of the `IntegralDomain` to permit the division. There are various axioms associated with this aspect of being an integral domain:

$$a * b = b * a$$

$$a, b \neq 0 \Rightarrow a * b \neq 0$$

$$b \neq 0 \wedge a \,\texttt{exquo}\, b \neq \texttt{"failed"} \Rightarrow b * (a \,\texttt{exquo}\, b) = b$$

$$a = b * c \Rightarrow a \,\texttt{exquo}\, b \neq \texttt{"failed"}.$$

---

* This is discussed by Davenport & Padget [1985a,b] and by Abbott *et al.* [1985]. The latter introduced the concept of "early abort" trial division.

As we remarked earlier, there is no very good reason why we have forced all integral domains to be commutative: it is just that we haven't seen any need for a category of non-commutative integral domains with `exquo`. It would certainly not be difficult to add such a category, but one would have to be careful as to whether one meant left-division or right-division. Whether or not this is done, the `exquo` operator is quite powerful.

Proposition. *In any* `IntegralDomain`*, it is possible to determine if two elements are associates or not.*

The proof is obvious.

But, from a computational point of view, there's more to being an integral domain than the existence of the `exquo` operator. We have already discussed the importance of *functoriality* in the abstract and here we have a good concrete example: there ought to be a functor `Fraction`, taking any `IntegralDomain` into its field of fractions (we describe this functor later). The obvious representation for such a functor is to represent a fraction by its numerator and denominator. What would it mean for this field of fractions to be canonically represented?

(1) The `IntegralDomain` itself must clearly be canonically represented.
(2) We must be able to suppress common divisors from the numerator and denominator of a fraction. This question is discussed in the next section: "Greatest Common Divisors".
(3) We must be able to choose which associate of the denominator to use, since a fraction is the same whatever unit we multiply the numerator and denominator by.

It is this last point that concerns us for the moment. We will require some form of operator which returns a distinguished associate of any element. Such operators are sometimes easy to find, and sometimes very difficult. For example, the normal choice for the integers would be "absolute value", and the normal choice for a polynomial domain is to ensure that the leading coefficient is, recursively, canonical. For the Gaussian integers we could choose one quadrant of the Argand diagram, say $x > 0, y \geq 0$.

Theorem. *There exist integral domains such that there cannot exist an algorithm for computing a canonical associate of every element.*

Proof. As Fröhlich & Shepherdson [1956] did, we construct a domain D which may be either $\mathbf{Z}$ or $\mathbf{Z}[\sqrt{2}, 1/\sqrt{2}]$. If it is $\mathbf{Z}$, then 1 and 2 are not associates, and the canonical form for 2 must be ± 2. If, however, it is $\mathbf{Z}[\sqrt{2}, 1/\sqrt{2}]$, then 1 and 2 are associates, and so must have the same canonical form. Asking the question "do 1 and 2 have the same canonical form" is equivalent to deciding on the nature of D, which is impossible.

We note that D is not an `IntegralDomain` in the Scratchpad sense, since it doesn't possess an `exquo` algorithm either, since knowing the value of $1 \,\texttt{exquo}\, 2$ would determine D. In fact, using Brown's trick [Brown, 1969], we can equip any `IntegralDomain` with a canonical associate operator: we keep a list of every canonical element encountered, and, every time the "canonical associate" question is asked of x, we return the first element of this list which is an associate of x. If there is no such element, x is deemed to be canonical, and is added to the list of canonical elements. Algorithms such as this, while in some sense they work, are not to be regarded favourably: partly because of their expense, but also because of their fundamentally non-canonical nature — organising a calculation in a different way, or a different choice of random numbers, can change the definition of "canonical", which is not particularly helpful.

Hence, from the point of view of efficient algorithms, we can see that some `IntegralDomain`s will have efficient algorithms for finding canonical associates, and some will not. It turns out to be more practical in Scratchpad to say that all `IntegralDomain`s should have an operator `canonical`, which always satisfies the axiom

$$x, \texttt{canonical}(x) \text{ are associates,}$$

but that the truly canonical nature of this, viz. that

$$x \text{ and } y \text{ are associates} \Rightarrow \texttt{canonical}(x) = \texttt{canonical}(y)$$

should be optional — if we know that this holds in a particular domain, we declare the attribute `canonicalUnitNormal` in that domain.

There is an additional question that has to be considered here: are the canonical elements closed under multiplication? This can be expressed axiomatically in the following way:

$$\texttt{canonical}(\texttt{canonical}(x) * \texttt{canonical}(y)) = \texttt{canonical}(x) * \texttt{canonical}(y).$$

Some domains have this property, e.g. the integers with the usual definition of `canonical` as "absolute value". The Gaussian integers don't have this property with the choice of a quadrant of the Argand diagram, but it is possible to find definitions of `canonical` which do have this property — choose, once and for all, a canonical associate for each prime of the Gaussian integers (for example, this could be in a particular quadrant), and then define the canonical associate of an arbitrary element to be the product of the canonical associates of its prime factors. This set of canonical associates is then closed under multiplication, but the algorithm for finding them is hardly efficient. Hence the axiom mentioned above is given a name — `canonicalsClosed`, and some domains assert its validity, while others don't. It is only asserted in the presence of `canonicalUnitNormal`. It is a consequence of this axiom that

$$x \,\texttt{exquo}\, y \neq \texttt{"failed"} \Rightarrow \texttt{canonical}(x) \,\texttt{exquo}\, \texttt{canonical}(y) \text{ is canonical.}$$

Proof. Let $z = \texttt{canonical}(x) \,\texttt{exquo}\, \texttt{canonical}(y)$. Since $z * \texttt{canonical}(y) = \texttt{canonical}(x)$, $z * y$ is an associate of x. Hence

$$\begin{aligned}\texttt{canonical}(x) = \texttt{canonical}(z * y) &= \texttt{canonical}(\texttt{canonical}(z) * \texttt{canonical}(y)) \\ &= \texttt{canonical}(z) * \texttt{canonical}(y),\end{aligned}$$

so $\texttt{canonical}(x) \,\texttt{exquo}\, \texttt{canonical}(y) = \texttt{canonical}(z)$. Since the result of the `exquo` operator is unique, $z = \texttt{canonical}(z)$.

Greatest Common Divisors

As was explained in the introduction, we have to distinguish between the existence of algorithms for the computation of greatest common divisors and the existence of algorithms for the computation of unique factorisation. A `GcdDomain` is defined to be an `IntegralDomain` with an additional operator

$$\gcd : \$ \times \$ \mapsto \$$$

satisfying the following axioms:

$$\begin{aligned} x \,\texttt{exquo}\, \gcd(x, y) &\neq \texttt{"failed"} \\ y \,\texttt{exquo}\, \gcd(x, y) &\neq \texttt{"failed"} \\ x \,\texttt{exquo}\, z \neq \texttt{"failed"} \wedge y \,\texttt{exquo}\, z &\neq \texttt{"failed"} \Rightarrow \gcd(x, y) \,\texttt{exquo}\, z \neq \texttt{"failed"} \\ \texttt{canonicalUnitNormal} &\Rightarrow \gcd(x, y) = \texttt{canonical}(\gcd(x, y)). \end{aligned}$$

It is a consequence of these axioms that $\gcd(x, y)$ and $\gcd(y, x)$ are associates, and hence that

$$\texttt{canonicalUnitNormal} \Rightarrow \gcd(x, y) = \gcd(y, x)$$

but this condition is not imposed more generally, since without `canonicalUnitNormal`, it is hard to ensure that the correct associate of the gcd has been found. This question can be seen as another illustration of the importance of associates in the constructive multiplicative theory.

It follows from the classical theory that, in a `GcdDomain`, factorisations into irreducible elements are unique (up to order and up to choice of associates). Such factorisations will exist if the domain is Noetherian, but we have not found any useful algorithmic categorisation* of "Noetherian". There is an attribute `Noetherian`, which is asserted by some domains, and propagated by some functors (e.g. `SparseUnivariatePolynomial`).

---

* One could imagine an operator `increase` which, given an ideal, either returned a larger ideal, or the word `"failed"`, indicating that the ideal was maximal. The axiom of Noetherianity would

The Functor `Fraction`

We are now able to describe the structure of the functor `Fraction`. The declaration of `Fraction` requires an `IntegralDomain` D as input, and essentially returns a `Field`. The representation chosen is that of an ordered pair: numerator and denominator. If `x` belongs to the quotient field then these are referred to within the functor as `x.num` and `x.den`: conversely, if n and d are two elements of D, then the fraction n/d in `$` is denoted `[n,d]`. In fact, operations for accessing these components, known as `numer` and `denom` are exported. This is in fact the triumph of pragmatism over purism, since these are not necessarily algebraic operations (in the sense of `$` being `canonical`). By this we mean that $a = b$ does not necessarily imply that `numer`$(a) =$ `numer`(b), as can be seen from the Scratchpad example quoted in Davenport & Trager [1990].

If D is a `GcdDomain`, then one defines auxiliary functions `cancelGcd` and `normalize`, both with signature `$` $\mapsto$ `$`. The first makes use of the `gcd` operation, while the second ensures that the denominator is `canonical` (of course, if the attribute `canonicalUnitNormal` is not present, this doesn't mean very much). Both operators update their argument, and return it as result. The propagation of `canonical` is dealt with by a clause:

```
if D has canonical and D has GcdDomain and D has canonicalUnitNormal
   then $ has canonical
```

in the declaration of the functor.

The basic arithmetic operators come in two varieties: for domains which aren't `GcdDomain`s, and for those which are. The definitions of the second variety cancel common divisors and use the `normalize` function. These are stated as separate operations, since it is generally more efficient to cancel greatest common divisors during an operation, rather than at the end, while this is not true for normalisation. Thus multiplication for `GcdDomain`s is defined as

```
x * y ==
        xx := [x.num,y.den]
        yy := [y.num,x.den]
        cancelGcd xx
        cancelGcd yy
        normalize [xx.num*yy.num,xx.den*yy.den]
```

making two small gcd calculations rather than one large one. The `canonicalsClosed` attribute would render the call to `normalize` unnecessary, but the extra complexity of a further set of conditional definitions seems too high for the small gain in run-time efficiency. Of course, this decision could be changed at any time just by changing the code of the functor `Fraction`.

The rest of the arithmetic operations are not worth considering in detail, but there are a couple of operations whose definitions are worth looking at. The first is $=$, defined as

```
x = y == x.num = y.num & x.den = y.den
```

if D is a `GcdDomain` with `canonicalUnitNormal`, otherwise* as

```
x = y == x.num * y.den = y.num * x.den
```

---

then translate into the assertion that the loop

while $I \neq$ `"failed"` **do**
$I :=$ `increase`(I)

always terminates (at least if I isn't the whole domain). However, this requires the introduction of "ideal" as a type (one might restrict oneself to finite-generated ideals from the point of view of representation, though the axiom should certainly apply to infinitely-generated ideals), and it's not clear how to turn this and the `GcdDomain` properties into an efficient algorithm for the factorisation of *elements*, rather than *ideals*. Of course, there is no problem in principal ideal domains.

* It could be argued that we should find a better default definition, since doing two large multiplications may well be unnecessary. For example, we could verify things like degree compatibility if the underlying domain were a polynomial domain.

The second is the operator `retractIfCan` with signature $\$ \mapsto$ `Union(D, "failed")`, satisfying the axiom

$$x = \frac{n}{1}, n \in D \Leftrightarrow \texttt{retractIfCan}(x) = n.$$

If D is a `GcdDomain` with `canonicalUnitNormal`, the definition is

```
retractIfCan x == if x.den = 1 then x.num
                                else "failed"
```

If D is a `GcdDomain`, but without `canonicalUnitNormal`, the definition is

```
retractIfCan x == (z:=recip x.den) case "failed"=> "failed"
                  z * x.num
```

If D is not a `GcdDomain`, then the definition is

```
retractIfCan x == x.num exquo x.den
```

Unique Factorisation Domains

If R is any `IntegralDomain`, we define the functor `Factored` to map R onto another structure, which can be viewed as "partially-factored elements of R". We then define a `UniqueFactorisationDomain` to be a `GcdDomain` with the following additional operators:

$$\begin{aligned} \texttt{prime} &: \$ \mapsto \texttt{Boolean} \\ \texttt{squareFree} &: \$ \mapsto \texttt{Factored}(\$) \\ \texttt{factor} &: \$ \mapsto \texttt{Factored}(\$), \end{aligned}$$

satisfying the obvious axioms, viz. that `prime` is true only if the element is prime (in the sense that it is not a unit, but any factorisation of it must contain a unit), `squareFree` and `factor` return elements with the same value, containing relatively prime square-free factors in the first case, and non-associate prime factors in the second, with the additional proviso that, if $ has the `canonicalUnitNormal` attribute, then the factors are canonical.

Mathematically speaking, the operator `factor` would suffice, since `prime` could test whether the result of factoring its argument had length 1 or not, and `squareFree` could call `factor` and then regroup all factors having the same multiplicity. But this would be over-kill. It might also seem surprising that `squareFree` is not defined earlier: surely for polynomial domains (though not for the integers) this is equivalent to the computation of greatest common divisors. Regrettably, this is not true for two reasons: the first is that, for polynomials over a ring, we should compute the square-free decomposition of the content as well as of the primitive part, and this is not necessarily equivalent to greatest common divisor calculations. The second is that, even for polynomials over a ring, the problem of computing square-free decomposition may be insoluble, even though greatest common divisors can be computed.

Proof. Let K be $(\mathbf{Z}/p\mathbf{Z})[y]$. As Fröhlich and Shepherdson [1956] did, we construct a domain L which might be K, or might be $K[y^{1/p}]$, and the consider the factorisation of $x^p - y$ in $L[x]$. If L is K, this is irreducible, and *a fortiori* square-free. The other possibility is that this factors as $\left(x - y^{1/p}\right)^p$, in which case it is not square-free.

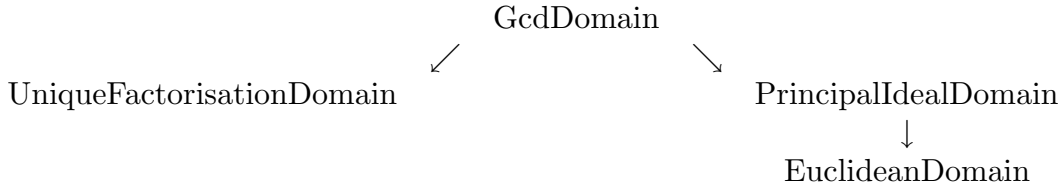
Euclidean Domains

In the normal development of commutative algebra, the sequence of refinement goes

$$\text{Unique Factorisation Domain} \longrightarrow \text{Principal Ideal Domain} \longrightarrow \text{Euclidean Domain}.$$

In practice, there are few examples of principal ideal domains which are not Euclidean domains, and in fact there are no such domains in Scratchpad.

The example of Fröhlich & Shepherdson shows that it is possible to have constructive Euclidean domains which are not constructive unique factorisation domains, so that our hierarchy will look like



`PrincipalIdealDomain` is defined to be a `GcdDomain` with an operator `principalIdeal` which, given a list of elements, finds a generator of the ideal they define This only ensures that finitely generated ideals are principal, but there are (non-constructive) axioms asserting that all ideals are principal.

The conventional definition of a Euclidean domain involves a function ϕ from the domain to some ordered (abelian) monoid, with the property that

$$x, y \neq 0 \Rightarrow \phi(xy) \geq \phi(x), \phi(y).$$

If one declares that $\phi(0)$ is not defined, one can regard the non-negative integers as the range of ϕ. `EuclideanDomain`s are defined to be extensions of `GcdDomain` with two additional fundamental operators:

$$\texttt{euclideanSize} : \$ \mapsto \texttt{NonNegativeInteger}$$
$$\texttt{div} : \$ \times \$ \mapsto \$ \times \$,$$

where `div` is infix, and the two components of the return type are called `quotient` and `remainder`. These satisfy the axioms:

$y \neq 0 \Rightarrow x = y * (x \,\texttt{div}\, y).\texttt{quotient} + (x \,\texttt{div}\, y).\texttt{remainder}$

$y \neq 0 \Rightarrow (x \,\texttt{div}\, y).\texttt{remainder} = 0 \vee \texttt{euclideanSize}((x \,\texttt{div}\, y).\texttt{remainder}) < \texttt{euclideanSize}(y)$

$x, y \neq 0 \Rightarrow \texttt{euclideanSize}(xy) \geq \texttt{euclideanSize}(y)$,

In such a domain, there are obvious default definitions for the functions `exquo`* and gcd:

```
x exquo y ==
   qr:=x div y
   qr.remainder = 0 => qr.quotient
   "failed"
```

and

```
gcd(x,y) ==
   x:=canonical(x)
   y:=canonical(y)
   while y ^= 0 repeat
      (x,y) = (y,(x div y).remainder)
      y:=canonical(y)
   x
```

Proposition. *This algorithm does in fact compute the greatest common divisor of its inputs.*

Proof. The partial correctness (i.e. the fact that, if the algorithm terminates, then it computes the correct result) of the algorithm follows exactly as in the classical case. If z is a common

* As was remarked earlier, this may well not be the most efficient definition for `exquo`.

divisor, then it divides x and y initially, and hence it divides x and y throughout the running of the algorithm, and in particular it divides the final value of x, which is the result. On the other hand, the result divides the last pair (x, y) (since $y = 0$). But each pair is a linear combination of the elements of the next pair, so by induction, the result divides the elements of every pair.

Hence we need merely show that the algorithm terminates, which is obvious since $\phi(y)$ is strictly decreasing. Furthermore, the guard $y \neq 0$ ensures that the division always succeeds.

Inside `EuclideanDomain`, we can give `principalIdeal` a default definition in terms of the extended Euclidean algorithm.

The Functor `SparseUnivariatePolynomial`

We are now able to describe the structure of the functor `SparseUnivariatePolynomial`. This is defined to take as parameter a ring R, and the return the ring of polynomials in one "anonymous" variable over this ring. In fact, the return type is not simply a ring, rather it is at least a `Ring` and an `Module` over R, with various other properties:

(1) If R is an `IntegralDomain`, then so is $;
(2) If R is a `GcdDomain`, then so is $;
(3) If R is a `Field`, then $ is a `EuclideanDomain`;
(4) If R has the `canonicalUnitNormal` attribute, then so does $;
(5) If R has the `canonicalsClosed` attribute, then so does $;
(6) If R is a `CommutativeRing`, the so is $, which is also an `Algebra` over R.
(7) If R has the `canonical` attribute, then so does $;
(8) If R has the `Noetherian` attribute, then so does $;

The representation chosen is that of a `List` of objects called `Terms`, each of which is a record with a component from R (known as `c`) and a non-negative integer (known as `k`). In the terminology of Stoutemyer [1984], the representation is sparse, and implicit in variables. Of course, R could itself be the result of calling `SparseUnivariatePolynomial`, so the representation is also capable of being recursive. Given this representation, most of the algorithms are fairly obvious (though a little care has to be taken, since it is not assumed that R is always commutative): the important point for this paper is to note how the correct properties of R let us define the correct operations for $.

For example, the function `canonical` for $ is defined to return 0 if the input is 0, otherwise $(\texttt{canonical}(\texttt{lc}(x))\ \texttt{exquo}\ \texttt{lc}(x)) * x$, where `lc` is the "leading coefficient" operator. Of course, this is not the only choice possible, but it is both natural and fairly efficient. It certainly does ensure the correctness of the propagation of the attribute `canonicalUnitNormal`, and indeed that of `canonicalsClosed`.

Conclusions

We see that, up to the category `IntegralDomain`, the conventional theory and the constructive theory are pretty much in step. When it comes to `IntegralDomain`, we have to convert a non-effective axiom into an operation, the uniqueness of whose result is guaranteed by the non-effective axiom. Every `IntegralDomain` can be extended to a quotient field, and the functor `Fraction` does precisely this. In order to get an efficient extension, and in particular to ensure that domains with the `canonical` attribute extend to fields with the `canonical` attribute, we require that the domain should be a `GcdDomain`, and that it should have the `canonicalUnitNormal` attribute. The first of these is fairly obvious, the second is a feature of the constructive theory. With these definitions, we have a general functor which has all the efficiency of the special cases "rational number" and "rational function" of traditional computer algebra systems, where this is possible.

From the constructive point of view, the categories `GcdDomain` and `UniqueFactorisationDomain` are very different. This is partly due to the fundamental difference between the operations: `gcd` depends only on its inputs (at least up to the choice of associates), whereas `factor` depends also on the ambient domain, and, as the example of Fröhlich and Shepherdson shows, this difference is crucial when it comes to questions of effectivity. The difference is also partly due to the fact

that we do not have an effective formulation of "Noetherian". We can formulate this as a question for future research:

- Does "Noetherian" have a useful constructive definition?

The major difference from the classical theory follows from the previous paragraph: a `EuclideanDomain` is not necessarily a `UniqueFactorisationDomain`. With this, we can build a successful abstract functor `SparseUnivariatePolynomial`, which models the classical theory, with one significant exception.

The classical theorem *polynomials over a unique factorisation domain form a unique factorisation domain* has no part in the constructive theory we have elaborated. There are two obvious reasons for this. The first is that it is false: in the Fröhlich–Shepherdson example, K is a field, hence a `UniqueFactorisationDomain`, but $K[x]$ cannot be a `UniqueFactorisationDomain`. The second is that it is unreasonable: the efficient algorithms that we know for factoring polynomials over the integers don't rely on the factorisation of integers (unless one insists that the content be completely factored), but do rely on other properties of polynomials over the integers (reduction modulo p; Hensel's Lemma) which our formulation does not capture at all. We can set this as a future research topic:

- Find a formulation of "unique factorisation" such that polynomials over a unique factorisation domain become a unique factorisation domain. It may be useful to consider condition (F) of Seidenberg [1974] in this context.

Acknowledgements.

Both authors are grateful to many past and present members of the Scratchpad group for their input to the theory described in this paper. Many of the orignial ideas were worked out in conjunction with D.R. Barton. The stimulus for writing the first version of this paper was provided by the Computer Algebra Group of Nice/Antipolis. Much discussion of this material took place while the authors enjoyed the hospitality of Mrs. Barbara Gatje.

References

[Abbott *et al.*, 1985] Abbott, J.A., Bradford, R.J. & Davenport, J.H., A Remark on Factorisation. SIGSAM Bulletin **19** (1985) 2, pp. 31–33, 37.

[Bareiss, 1968] Bareiss, E.H., Sylvester's Identity and Multistep Integer-preserving Gaussian Elimination. Math. Comp. **22** (1968) pp. 565–578.

[Bradford *et al.*, 1986] Bradford, R.J., Hearn, A.C., Padget, J.A. & Schrüfer, E., Enlarging the REDUCE Domain of Computation. Proc. SYMSAC 86 (ACM, New York, 1986) pp. 100–106.

[Brown, 1969] Brown, W.S., Rational Exponential Expressions, and a conjecture concerning π and e. Amer. Math. Monthly **76** (1969) pp. 28–34.

[Collins, 1966] Collins, G.E., PM, a system for polynomial multiplication. Comm. ACM **9** (1969) pp. 578–589.

[Davenport, 1981a] Davenport, J.H., On the Integration of Algebraic Functions. Springer Lecture Notes in Computer Science 102, Springer-Verlag, Berlin-Heidelberg-New York, 1981 [Russian ed. MIR, Moscow, 1985].

[Davenport, 1981b] Davenport, J.H., Effective Mathematics — the Computer Algebra viewpoint. Proc. Constructive Mathematics Conference 1980 (ed. F. Richman) [Springer Lecture Notes in Mathematics 873, Springer-Verlag, Berlin-Heidelberg-New York, 1981], pp. 31–43.

[Davenport & Padget, 1985a] Davenport, J.H. & Padget, J.A., HEUGCD: How Elementary Upperbounds Generate Cheaper Data. Proc. EUROCAL 85, Vol. 2 (Springer Lecture Notes in Computer Science 204, Springer-Verlag, Berlin-Heidelberg-New York, 1985) pp. 18–28

[Davenport & Padget, 1985b] Davenport, J.H. & Padget, J.A., On Numbers & Polynomials. Computers and Computing (ed. P. Chenin, C. Dicrescenzo, F. Robert), Masson and Wiley, 1985, pp. 49–53.

[Davenport & Trager, 1990] Davenport, J.H. & Trager, B.M., Scratchpad's View of Algebra I: Commutative Algebra. IBM Research Report RC 14897 and University of Bath Computer Science Technical Report 90-31, January 1990.

[Fröhlich & Shepherdson, 1956] Fröhlich, A. & Shepherdson, J.C., Effective Procedures in Field Theory. Phil. Trans. Roy. Soc. Ser. A **248** (1955–6) pp. 407–432.

[Gianni *et al.*, 1988] Gianni, P., Trager, B.M. & Zacharias,G., Gröbner Bases and Primary Decomposition of Polynomial Ideals. J. Symbolic Comp. **6** (1988) pp. 149–167.

[Hearn, 1979] Hearn,A.C., Non-Modular Computation of Polynomial Gcd Using Trial Division. Proc. EUROSAM 79 (Springer Lecture Notes in Computer Science 72, Springer-Verlag, Berlin-Heidelberg-New York) pp. 227–239.

[Hermite, 1872] Hermite, E., Sur l'intégration des fractions rationelles. Nouvelles Annales de Mathématiques, 2 Sér., **11** (1872) pp. 145–148. Ann. Scientifiques de l'École Normale Supérieure, 2 Sér., **1** (1872) pp. 215–218.

[Jenks & Trager, 1981] Jenks, R.D. & Trager, B.M., A Language for Computational Algebra. Proc. SYMSAC 81 (ACM, New York, 1981) pp. 6–13. Reprinted in SIGPLAN Notices **16** (1981) No. 11, pp. 22–29.

[Moore & Norman, 1981] Moore, P.M.A. & Norman, A.C., Implementing a Polynomial Factorization and GCD Package. Proc. SYMSAC 81 (ACM, New York, 1981) pp. 109–116.

[Seidenberg, 1974] Seidenberg, A., Constructions in Algebra. Trans. AMS **197** (1974) pp. 273–313.

[Stoutemyer, 1984] Stoutemyer, D.R., Which Polynomial Representation is Best: Surprises Abound. Proc. 1984 MACSYMA Users' Conference (ed. V.E. Golden), G.E., Schenectady, pp. 221–243.

[Trager, 1976] Trager,B.M., Algebraic Factoring and Rational Function Integration. Proc. SYMSAC 76 (ACM, New York, 1976) pp. 219–226.