Scratchpad's View of Algebra II: A Categorical View of Factorization

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

P.Gianni
Dipartimento di Matematica
Università di Pisa
Via Buonarroti 2
56100 Pisa
Italy

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

Abstract
This paper explains how Scratchpad solves the problem of presenting a categorical view of factorization in unique factorization domains, i.e. a view which can be propagated by functors such as `SparseUnivariatePolynomial` or `Fraction`. This is not easy, as the constructive version of the classical concept of `UniqueFactorizationDomain` cannot be so propagated. The solution adopted is based largely on Seidenberg's conditions (F) and (P), but there are several additional points that have to be borne in mind to produce reasonably efficient algorithms in the required generality.

The consequence of the algorithms and interfaces presented in this paper is that Scratchpad can factorize in any extension of the integers or finite fields by any combination of polynomial, fraction and algebraic extensions: a capability far more general than any other computer algebra system possesses. The solution is not perfect: for example we cannot use these general constructions to factorize polynomials in $\overline{\mathbf{Z}[\sqrt{-5}]}[x]$ since the domain $\mathbf{Z}[\sqrt{-5}]$ is not a unique factorization domain, even though $\overline{\mathbf{Z}[\sqrt{-5}]}$ is, since it is a field. Of course, we can factor polynomials in $\overline{\mathbf{Z}}[\sqrt{-5}][x]$.

1. Introduction

Scratchpad [Jenks *et al.*, 1988] is a computer algebra system based on the "abstract data type" view of computing, where the various data types are constructed from elementary ones, such as `Integer`, by means of *functors* (functions which return types) such as `SparseUnivariatePolynomial` or `Fraction`. The types consist of a data representation and operations, so that a type such as `List(Integer)` provides the operations `length` etc. as well as the data structure of a list. The types are themselves typed, with second-order types being called *categories*. These categories are defined to supply certain operations, so that every `Ring` will provide operations such as addition and multiplication. The functors define how each operation is implemented for the data types they return. The choice of definition can depend on the properties of the parameters, so that an exponentiation for polynomials over `R` which was implemented via the binomial theorem for commutative coefficients, but by repeated multiplication otherwise, can be easily defined, as in the actual Scratchpad code

```
if R has CommutativeRing then
  p ** nn  ==
     null p => 0
     nn = 0 => 1
     p.rest = [] =>
       [[nn * p.first.k, p.first.c ** nn]]
     binomThmExpt([p.first], p.rest, nn)
 else
  p ** nn  == repeatMultExpt(p,nn)
```

where the definitions of `binomThmExpt` and `repeatMultExpt` have been omitted for brevity.

A previous paper [Davenport & Trager, 1990] explained the basis of Scratchpad's view of commutative algebra. In that paper we explained why a constructive view of factorization is inherently different from a non-constructive one, as originally pointed out by Fröhlich & Shepherdson [1956]. Hence Scratchpad has to distinguish between the categories of `GcdDomain`, viz. an integral domain in which greatest common divisors can be computed, and `UniqueFactorizationDomain`, viz. an integral domain in which, additionally, factorizations into irreducible elements can be computed (and therefore are unique up to order and choice of associates). Every `UniqueFactorizationDomain` is a `GcdDomain`, but the converse does not hold.

Most of the categories of Scratchpad are *functorial*, in the sense that functors such as `SparseUnivariatePolynomial` preserve the property of belonging to this category when appropriate. For example, the theorem "if R is an integral domain, so is $R[x]$" is translated into the constructive formulation of Scratchpad by saying that the functor `SparseUnivariatePolynomial` preserves the category `IntegralDomain`, or that `Inte-`

gralDomain is functorial for SparseUnivariatePolynomial, and by writing

```
if R has IntegralDomain then IntegralDomain
```

in the definition of SparseUnivariatePolynomial. Fröhlich & Shepherdson [1956] pointed out that UniqueFactorizationDomain cannot be functorial for SparseUnivariatePolynomial, i.e. there can be no way for Scratchpad to implement the theorem "if R is a unique factorization domain, so is $R[x]$", and Davenport & Trager [1990] concluded by posing the problem of finding an appropriate formulation of factorization which would be functorial.

It is the aim of this paper to explain how this problem has been resolved in the latest versions of Scratchpad. We consider in the next section the problem of domains of characteristic zero (i.e. belonging to the category CharacteristicZero), and then go on to the more difficult problem of domains of non-zero characteristic (i.e. belonging to the category CharacteristicNonZero).

2. Characteristic Zero

The categorical structure is largely inspired by the treatment in Seidenberg [1974] of "condition (F)", which he defined as the ability to construct the complete factorization of polynomials on one variable over a field k. We say that a domain R (which must already be an UniqueFactorizationDomain) belongs to the category PolynomialFactorizationExplicit (in characteristic zero — see section 4 for additional properties in finite characteristic) if it also has the following additional operations, where signatures are represented in the usual Scratchpad notation, and P represents the domain of polynomials in an (anonymous) variable over R — Scratchpad's SparseUnivariatePolynomial(R).

```
squareFreePolynomial: P -> Factored(P)
factorPolynomial: P -> Factored(P)
factorSquareFreePolynomial: P -> Factored(P)
gcdPolynomial: (P, P) -> P
solveLinearPolynomialEquation:
      (List P, P) -> Union(List P,"failed")
```

We will consistently use x as the denotation for this anonymous variable — this is purely an expository convenience, and the reader must remember that Scratchpad has no difficulty in handling domains containing several instantiations of SparseUnivariatePolynomial, since the type structure makes it clear at any moment which instantiation is being talked about.

The operation factorPolynomial corresponds explicitly to Seidenberg's condition (F). The operations squareFreePolynomial, which returns a square-free factorization (with respect to the anonymous variable) of a given polynomial, and factorSquareFreePolynomial, which produces the complete factorization of a polynomial already known to be non-trivial and both square-free and primitive *with respect to the anonymous variable*, are included for efficiency (in the case of CharacteristicZero — see later for the treatment of inseparability in finite characteristic). In fact, factorPolynomial can be defined in terms of these other two, though a little care has be taken with the content with respect to the anonymous variable.

The operation gcdPolynomial, which computes the greatest common divisor of two polynomials in the anonymous variable, is logically redundant, in fact for two different reasons. The first is that, since R is a UniqueFactorizationDomain, it is *a fortiori* a GcdDomain, and therefore P is also a GcdDomain. The second reason is that greatest common divisors can be computed by collecting the common terms in the factorizations of the two polynomials. Nevertheless, this operation is of substantial practical importance — it will be shown that this formulation of the problem allows us to use efficient modular or p-adic methods for the computation of greatest common divisors, rather than the subresultant method which is all that the functoriality of GcdDomain allows us. There is a *default* definition provided via the subresultant algorithm [Loos, 1982], but most constructors which yield PolynomialFactorizationExplicit types actually provide their own.

The operation solveLinearPolynomialEquation is less obvious, though we will see that it is crucial to our formulation of Hensel's Lemma. Given a set of polynomials (in the anonymous variable) $f_1, \ldots, f_n$ which have no common factor, and a polynomial g, it returns either a list of polynomials $a_1, \ldots, a_n$ such that $\sum a_i \prod_{j \neq i} f_j = g$ and a_i has degree strictly less than f_i, or the token failed if no such set exists. This degree constraint ensures that the answer is unique, and that the answer is useful in Hensel lifting. We note that there is a possibility for failure: we have ensured that g is divisible by the common divisor of the f_i, but it is possible for no solutions to exist in $R[x]$ even though they do exist in $k[x]$ where k is the field of fractions of R. For example, consider

```
solveLinearPolynomialEquation([x-1,x+1],1)
```

(x being the anonymous variable) which would return failed over $\mathbf{Z}$, but $\left[\frac{-1}{2}, \frac{1}{2}\right]$ over $\mathbf{Q}$. If R is a Field, then $R[x]$ is a EuclideanDomain, and the operation solveLinearPolynomialEquation over R is identical to the operation multiEuclidean over $R[x]$, and indeed this is expressed via a conditional default. Otherwise, we can always extend R to its field of fractions, solve the problem there, and endeavour to retract, and this is also provided as a default operation. It is an implicit part of the characterization of solveLinearPolynomialEquation that the operation may often be called with the same first argument, as happens in linear Hensel lifting, and implementations are expected to do the appropriate caching.

Let us now examine the algorithms actually used for the various functors in Scratchpad which define or

propagate the category `PolynomialFactorizationExplicit`, restricting our attention for the moment to the case of characteristic zero, and discussing non-zero characteristic in later sections.

`Integer` — $\mathbf{Z}$

The algorithmic requirements to make this ring be `PolynomialFactorizationExplicit` are quite straightforward: the `factorSquareFreePolynomial` operation is provided by the classical scheme of reduction to a prime p, "distinct degree" factorization [Cantor & Zassenhaus, 1981], lifting to p^n and re-combination. The reduction to square-free polynomials is simple [Yun, 1976; 1977]. Although the default `gcdPolynomial` and `solveLinearPolynomialEquation` implementations would be perfectly satisfactory, better ones are available. For `gcdPolynomial` we use the "evaluate at a large integer" method [Char *et al.*, 1984] [Davenport & Padget, 1985a; 1985b], and for `solveLinearPolynomialEquation` we reduce modulo a word-sized prime (such that the system stays non-singular), and use Hensel's Lemma if necessary to obtain a large enough modulus, based on a Hadamard-style bound for the size of a possible solution. We need such a bound since solutions may not exist over $\mathbf{Z}$.

`Fraction` — field of fractions

To fix notation, let D be the argument to `Fraction`, which has to be an `IntegralDomain` in general, and which needs to be `PolynomialFactorizationExplicit` in order for factorization to be propagated, k be the result of `Fraction`, which is a `Field`, and R be $k[x]$, using x to stand for the anonymous variable.

The algorithm for `factorPolynomial` is conceptually straight-forward — clear fractions, factorize the polynomial over $D[x]$ (using a call to `factorPolynomial` from the domain D), and re-write the results to lie in $k[x]$. The algorithm for `factorSquareFreePolynomial` is identical except that it recurses through `factorSquareFreePolynomial` from D. This explains why we stipulated that the arguments for `factorSquareFreePolynomial` should be square-free and primitive *with respect to the anonymous variable only*. For example, let D be $\mathbf{Z}[y]$, then the polynomial $f = (y-1)^2(x^2-1)$ is square-free in $k[x]$, since $(y-1)^2$ is a unit in k, but f is not square-free or primitive in $\mathbf{Z}[y][x]$. However, it is legitimate to call `factorSquareFreePolynomial` on f in $D[x]$.

The operation `gcdPolynomial` works similarly — denominators are cleared from both inputs, and the operation `gcdPolynomial` from D is called. The result has to be made monic as a polynomial in $k[x]$, since this is the unit normalization for greatest common divisors assumed by `SparseUnivariatePolynomial`. The operation `solveLinearPolynomialEquation` takes its default value, viz. the `multiEuclidean` operation from R. It might be possible to write a better algorithm, but the point is that the system may well have solutions in $k[x]$ without having solutions in $D[x]$, so a simple reduction from k to D would be incorrect.

`SparseUnivariatePolynomial` — $[y]$

To fix notation, let R be the first argument to `SparseUnivariatePolynomial`, viz. a `PolynomialFactorizationExplicit` ring, and let y be the name of the variable in which we are making polynomials, reserving x for the anonymous variable over which the operations defining `PolynomialFactorizationExplicit` are stated.

The algorithms for factorization work by reducing the problem from $R[y][x]$ to $R[x]$, solving the problem there since R is `PolynomialFactorizationExplicit`, and lifting the results.

Let us look first at applying `factorSquareFreePolynomial` to a polynomial f. The general strategy is simple. Write l for $\mathrm{lc}_x(f)$.

[1] Choose a value a in R for y such that $f(a,x)$ is square-free and has the same degree as f (i.e. any value such that $y-a$ does not divide $\mathrm{lc}_x(f)\mathrm{disc}_x(f)$).

[2] Remove the content from $f(a,x)$ — call the result $g(x) \in R[x]$.

[3] Call `factorSquareFreePolynomial` from R on g, which is justified since a was chosen to make g square-free and non-trivial, and g is primitive.

[4] If g is irreducible, then f is irreducible, and we terminate by returning a one-element factorization. Otherwise, let n be the number of factors found.

[5] Multiply one of the factors found by $f(a,x)/g$ and impose l as the leading coefficient of each factor. Call these factors $g_1, \ldots, g_n$. Write $F = f(x,y)l^{n-1}$ so that $F \equiv \prod g_i \pmod{y-a}$.

[6.1] Write

$$E = \left.\frac{F - \prod g_i}{y-a}\right|_{y=a},$$

and call `solveLinearPolynomialEquation` (from the domain R) on the g_i and E. If this returns `failed`, then the g_i are not images of a factorization of F and we go to [7], otherwise it returns a list of polynomials a_i, and we replace g_i by $g_i + (y-a)a_i$.

[6.2] Write

$$E = \left.\frac{F - \prod g_i}{(y-a)^2}\right|_{y=a}.$$

If $E = 0$, we have a factorization of F, and we convert this to a factorization of f by making the g_i primitive (and taking care over units!). Otherwise we call `solveLinearPolynomialEquation` (from the domain R) on the g_i (which is the same argument as last time, and `solveLinearPolynomialEquation` implementations are expected to be efficient in such circumstances) and E. If this returns `failed`, then the g_i are not images of a factorization of F and we go to [7], otherwise it returns a list of polynomials a_i, and we replace g_i by $g_i + (y-a)^2 a_i$.

[...]

[6.m] Write

$$E = \left.\frac{F - \prod g_i}{(y-a)^m}\right|_{y=a}.$$

If $E = 0$, we have a factorization of F, and we convert this to a factorization of f by making the g_i primitive (and taking care over units!). Otherwise we call `solveLinearPolynomialEquation` (from the domain R) on the g_i and E. If this returns `failed`, then the g_i are not images of a factorization of F and we go to [7], otherwise it returns a list of polynomials a_i, and we replace g_i by $g_i + (y-a)^m a_i$.

Repeat these steps until the factorization is found, or until the total degree (in y) of $\prod g_i$ exceeds the degree of F, in which case we have a failure (caused by a false split).

[7] [This step is reached by fall-through from the previous loop, or by a failure of the calls to `solveLinearPolynomialEquation`.] Choose a different value a according to the criteria in [1], and restart.

The algorithm for `gcdPolynomial` works by reducing the problem from $R[y][x]$ to $R[x]$, solving the problem there since R is `PolynomialFactorizationExplicit`, and lifting the results as a two-element factorization, using `solveLinearPolynomialEquation` as in the lifting of factorizations. Since R is infinite, almost all reductions from $R[y][x]$ to $R[x]$ will be lucky, in the sense that the gcd in $R[x]$ is the image of the gcd in $R[y][x]$.

The algorithm for `solveLinearPolynomialEquation` works by reducing the problem from $R[y][x]$ to $R[x]$, (ensuring that the polynomials preserve their degree and that the list of polynomials stays relatively prime) solving the problem there since R is `PolynomialFactorizationExplicit`, and lifting the results.

`SimpleAlgebraicExtension` — $[\theta]$

Here the algorithm for `factorSquareFreePolynomial` is essentially that of Trager [1976]: we take norms from $R[\theta][x]$ to $R[x]$, ensuring that the result stays square-free (which can only fail at a finite number of linear substitutions of the form $\theta \mapsto \theta - a$), factor the norm over $R[x]$, which is possible since R is `PolynomialFactorizationExplicit`, and use greatest common divisors to find the factor of the original polynomial. We currently rely on the default implementations for `gcdPolynomial` and `solveLinearPolynomialEquation`.

3. Finite Characteristic — Condition F

The two major problems in extending the algorithms given above to the case of finite characteristic are that such fields may be *too small* and there exist non–constant polynomials with zero derivative. The first problem, that finite fields may not contain enough values, implies that a polynomial $f(y,x)$ may be square-free, but $\mathrm{disc}_y(f)$ may vanish at every element of the given finite field. To deal with this problem we may need to *grow* the field to contain sufficiently many *good* values. The other problem is equivalent to the problem of inseparability, and shows up as added complexity in the `squareFreePolynomial` operation and is dealt with in the next section. Throughout this and the next section, p will denote the characteristic of the fields being discussed.

`Finite Field Category` — Finite Fields

Finite fields can be constructed by the functor `PrimeField`, whose single argument is a prime number giving the size of the field to be constructed, or by `FiniteFieldExtension`, which, given a finite field and n, constructs the extension of that field of relative degree n. These are combined by the functor `FiniteField` — a call with arguments p and n generates the field with p^n elements.

The algorithmic requirements to make these fields belong to the `PolynomialFactorizationExplicit` category are quite straight-forward: the `factorSquareFreePolynomial` operation is provided by the "distinct degree" algorithm [Cantor & Zassenhaus, 1981], reduction to square-free is simple and the default `gcdPolynomial` and `solveLinearPolynomialEquation` implementations are perfectly satisfactory.

`SparseUnivariatePolynomial` — $[y]$

Let k be a member of `FiniteFieldCategory`. Factorization in $k[x]$ is possible, i.e. k satisfies Seidenberg's condition (F), and, as we have seen, is easily a member of the `PolynomialFactorizationExplicit` category. Factorization in $k[y][x]$ is also posssible (though not so often implemented in computer algebra systems), i.e. $k[y]$ satisfies condition (F). However, the algorithms given in the previous section will *not* make it a member of the `PolynomialFactorizationExplicit` category, since we may be unable to find a reduction from $k[y][x]$ to $k[x]$ preserving the square-free nature of our polynomial, and, even if we can, such reductions may not preserve the factorization structure of our polynomial — indeed they are unlikely to.

The onus is on `SparseUnivariatePolynomial` to ensure that $k[y]$ is a `PolynomialFactorizationExplicit`. This is achieved by special code in the definition of `SparseUnivariatePolynomial`. The `gcdPolynomial` operation is implemented via a subresultant method. The implementation of `solveLinearPolynomialEquation` reduces the problem to a univariate one, and then lifts, extending the field if no element of the original prime is suitable. `factorSquareFreePolynomial` determines *a priori* a large enough field extension and chooses evaluation points in this extension to ensure that the square-free polynomial remains square-free. If `R` has `FiniteFieldCategory` then we will need a step at the end of the lifting to combine factors until we find

true factors. Otherwise we assume that our specialization preserves the structure of the factorization and we only need to test divide the lifted factors.

`Fraction` — field of fractions

This constructor is handled the same way as in characteristic zero.

`SimpleAlgebraicExtension` — $[\theta]$

We will assume that the polynomial to factor is square-free, since the hard part of ensuring this is to do with inseparability, which is discussed in the next section. We must now distinguish two cases: either the ring over which we are taking an extension is finite, or it is not. If the ring is finite, then it is a finite field, and an algebraic extension of a finite field is itself a finite field. Hence the "distinct degree" algorithm [Cantor & Zassenhaus, 1981] is a suitable polynomial factorization algorithm. In the case of infinite fields, we will split our extension into a separable extension followed by a purely inseparable one. Let $f(z)$ be the minimal polynomial of θ over k. We first find the maximal value of r such that $f(z) = g(z^{p^r})$. $r = 0$ if and only if θ is separable over k. Thus we split our extension into a tower of two extensions. First we extend by α a root of $g(y)$ which gives a separable extension. Then we extend by $\theta = \alpha^{1/p^r}$, which gives a purely inseparable extension. To factor over separable extensions of infinite fields we can use Trager's [1976] algorithm as if the characteristic were zero. To factor $h(x)$ over a purely inseparable extension of degree p^r of some field k, we first factor the polynomial $H(x) = h(x)^{p^r}$ over k. Note that H is simply the norm of h and has all its cofficients in k. For each irreducible factor H_i of H we compute the $h_i = gcd(h, H_i)$ which yields the corresponding irreducible factor of h.

4. Finite Characteristic — Condition P

Throughout this section, p will denote the characteristic of the fields being discussed. The other problem that arises in finite characteristic is that of inseparability. We are used to the fact that the factorization of a polynomial depends on the ambient field, but in finite characteristic the square-free decomposition may also depend on the field. For example, the polynomial $x^p - y$ is either irreducible or a perfect p-th power, depending on whether or not the ambient field contains $y^{1/p}$. Seidenberg [1970] introduces an additional condition, which he calls (P), on fields to provide a constructive mechanism for handling this difficulty. He says that a field k satisfies condition (P) if, given a system of homogenous linear equations over k, we can decide if it has a non-trivial solution in k^p, and, if so, exhibit one.

We have chosen to extend `PolynomialFactorizationExplicit` to provide this functionality (logically speaking, we could define a separate category with this operation, but this seems unnecessary). Hence the definition of `PolynomialFactorizationExplicit` given in section 2 is augmented by

```
if $ has CharacteristicNonZero then
  conditionP: Matrix $ ->
              Union(Vector $,"failed")
  CharthRoot: $ -> Union($,"failed")
```

Here the operation `conditionP` corresponds to Seidenberg's definition generalized to rings — given a matrix over k, either return `failed` or a vector of elements of k. Seidenberg insisted that the solution should be in k^p, but algorithmically one wishes to know the precise expression of these solutions as elements of k^p, so the vector returned is the vector of p-th roots of a non-trivial solution of the homogenous linear equations implied by the matrix. The operation `CharthRoot` returns the p-th root of an element (or `failed` if there is no such root). It is logically not necessary, since it can be defined in terms of `conditionP`, but is provided for efficiency and as a convenience to the programmer.

`FiniteFieldCategory` — Finite Fields

The implementation of `CharthRoot` is simple enough in finite fields: in prime fields it is the identity, otherwise, in a field of size p^n, we raise elements to the power p^{n-1}. For `conditionP`, we first solve the linear system, and, if this is possible, we then take the p-th roots of the solution's elements.

`Fraction` — field of fractions

The implementation of `conditionP` is simple, we clear fractions from each equation, and then call `conditionP` from the domain whose fractions we are implementing. If this call succeeds, then we attach a denominator of 1 (by convention) to each element of the solution. For domains which are stored canonically, the implementation of `CharthRoot` is to compute, using `CharthRoot` in the domain whose fractions we are implementing, the p-th root of the numerator and of the denominator. The answer is then the quotient of the two p-th roots, and exists only if both p-th roots exists. If the domain is not stored canonically, we use the identity $\sqrt[p]{\frac{n}{d}} = \frac{1}{d}\sqrt[p]{nd^{p-1}}$ to reduce the problem to that of `CharthRoot` in the domain whose fractions we are implementing.

`SparseUnivariatePolynomial` — $[y]$

The implementation of `CharthRoot` relies on the result that a polynomial is a perfect p-th power if, and only if, it is a polynomial in y^p, each of whose coefficients is a perfect p-th power. Hence we need merely check the exponents, and call `CharthRoot` on the coefficients if appropriate. The same is essentially true of `conditionP`: all the polynomials occurring must have exponents a power of p, and the corresponding coefficients must be solutions of a system obtained by equating coefficients. This system can be solved by calling `conditionP`() from the underlying domain.

`SimpleAlgebraicExtension` — $[\theta]$

`SimpleAlgebraicExtension`s of elements of the category `FiniteFieldCategory` are themselves in `FiniteFieldCategory`, and these implementations have already been discussed. So we confine ourselves here to the case of applying `SimpleAlgebraicExtension` to an infinite field k of characteristic p.

For a purely separable extension generated by θ satisfying a minimal polynomial m of degree n, we write each unknown X over $k[\theta]$ as $X_0 + X_1\theta^p + \cdots + X_{n-1}\theta^{(n-1)p}$, similarly with the coefficients of the matrix, and reduce modulo m. This converts a `conditionP` problem over $k[\theta]$ to one with n times as many equations and unknowns over k, which is soluble if, and only if, the original system was soluble: if the p-th roots of the X_i are given as Y_i, then the p-th roots of the solution of the original system, i.e. the answer that `conditionP` should return, are of the form $\sum Y_i\theta^i$. A `CharthRoot` problem over $k[\theta]$ is converted into an $n \times n$ `conditionP` problem over k, which explains the importance of the more general `conditionP` procedure.

For a purely inseparable extension generated by θ satisfying a minimal polynomial $m(\theta) = \theta^p - a$, we write each unknown X over k as $X_0 + X_1 a + \cdots + X_{p-1}a^{p-1}$, similarly write the coefficients of the matrix as $c_0 + c_1\theta + \cdots + c_{p-1}\theta^{p-1}$, and equate coefficients of powers of θ. This converts a `conditionP` problem over $k[\theta]$ to one with p times as many equations and unknowns over k, which is soluble if, and only if, the original system was soluble. A `CharthRoot` problem over $k[\theta]$ is converted into n `CharthRoot` problems over k.

A mixed extension is treated as a combination of purely separable and purely inseparable extensions.

`squareFreePolynomial` — square-free factorization

Now that we have shown how to guarantee our ability to compute p-th roots, we can explain the necessary modifications to the `squareFreePolynomial` algorithm in finite characteristic. If we perform the same steps as we do in characteristic zero, we eventually arrive at a polynomial whose derivative with respect to x is zero. In characteristic zero this would mean that this polynomial no longer depended on x, but in finite characteristic, it simply means that $f(x) = g(x^p)$, i.e. our polynomial is in fact a polynomial in x^p. At this point we compute this polynomial $g(x)$ which is just f with all its exponents divided by p. Now we need to perform a complete factorization of $g(x)$. Thus in finite characteristic, the ability to do a square free factorization is intertwined with the ability to factor. To factor f we first need to square-free decompose f, but this requires us to factor g. We are making progress since the degree of g is smaller than the degree of f by at least a factor of p. If we let $\prod g_i^{a_i}(x)$ be the complete factorization of $g(x)$, then $f(x) = \prod g_i^{a_i}(x^p)$. There remains only to decide whether $g_i(x^p)$ is square-free or not. It is at this point in the factorization algorithm that we need to use the `CharthRoot` operation that we have just introduced. $g_i(x^p)$ is a p-th power if and only if its coefficients are p-th powers. We can decide this using `CharthRoot`.

5. Conclusions

We have presented the novel features of the algorithms that Scratchpad uses to implement its categorical view of factorization. The key concept is that of a `PolynomialFactorizationExplicit` domain, which, roughly speaking, is one that understands factorization of univariate polynomials over itself, including, in characteristic p, the factorization of $x^p - y$ for y in the domain.

We do not claim that the algorithms presented in this paper (which are complemented, in Scratchpad, by additional algorithms for common special cases) are the most efficient in any particular case, rather we claim that they are sufficiently efficient (unlike Seidenberg's [1974] treatment for example, where Kronecker's trick is used to reduce multivariate polynomial factorization to univariate factorization), and, more importantly, that they really exist in the generality claimed for them, so that, for example, Scratchpad can factor polynomials in the type $\overline{\mathbf{Q}[\sqrt{-2}][x][\sqrt{x}][y]}$ since the system will choose the appropriate algorithms and recursions based on the structure of this type, since $\mathbf{Q}$ has the appropriate properties, which are preserved by the functors $\sqrt{-2}$, $[x]$, $\overline{\text{(field of fractions)}}$ and $\sqrt{x}$.

There are several aspects of general algorithms for factorization which still need to be developed. We mentioned in the abstract one difficulty with our current formulation — all the intermediate domains need to be unique factorization domains. In the example quoted, $\overline{\mathbf{Z}[\sqrt{-5}]}[x]$ cannnot be deduced to be `PolynomialFactorizationExplicit`, since deductions about the `PolynomialFactorizationExplicit` of domains are made by functoriality, and $\mathbf{Z}[\sqrt{-5}]$ is not a unique factorization domain, so cannot be `PolynomialFactorizationExplicit`. If it were possible to weaken the definition of R being `PolynomialFactorizationExplicit`, so that the factorizations of elements of $P = R[x]$ involved did not have to unique in the traditional sense (unique up to units and order), but only in some weaker sense, such as unique up to elements of R, we might be able to factorize over a still wider class of rings.

Nothing in this paper has addressed the problems of factorization in non-commutative rings, or in differential rings, for the simple reason that the known algorithms are sufficiently different from those for the commutative case that synthesis appears to be impossible.

Acknowledgements. This paper has benefited from the contributions of many members of the Scratchpad group, and from IBM's support of all the authors. A longer version of this paper, with more details of the Scratchpad code, is available as University of Bath Computer Science Technical Report 91–46.

6. References

[Cantor & Zassenhaus, 1981] Cantor,D.G. & Zassenhaus,H., A New Algorithm for Factoring Polynomials over Finite Fields. Math. Comp. **36** (1981) pp. 587–592. Zbl. 493.12024. MR 82e:12020.

[Char *et al.*, 1984] Char,B.W., Geddes,K.O. & Gonnet, G.H., GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. Proc. EUROSAM 84 (Springer Lecture Notes in Computer Science 174, Springer-Verlag, 1984) pp. 285–296.

[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 Vol. 204, Springer-Verlag, 1987) 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: Basic Commutative Algebra. Proc. DISCO '90 (Springer Lecture Notes in Computer Science Vol. 429, 1990) pp. 40–54. A longer version appears as University of Bath Computer Science Technical Report 90-31.

[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. Zbl. 70,35.

[Jenks *et al.*, 1988] Jenks,R.D., Sutor,R.S. & Watt,S.M., Scratchpad II: An Abstract Datatype System for Mathematical Computations. Proc. "Trends in Computer Algebra" (Springer Lecture Notes in Computer Science 296, 1988) pp. 12–37.

[Loos, 1982] Loos,R., Generalized Polynomial Remainder Sequences. Symbolic & Algebraic Computation (Computing Supplementum 4) (ed. B. Buchberger, G.E. Collins & R. Loos) Springer-Verlag, Wien-New York, 1982, pp. 115–137

[Seidenberg, 1970] Seidenberg,A., Construction of the Integral Closure of a Finite Integral Domain. Rend. Sem. Mat. Fis. Milano **40** (1970) pp. 100–120. MR 45 #3396.

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

[Trager, 1976] Trager,B.M., Algebraic Factoring and Rational Function Integration. Proc. SYMSAC 76 (ACM, New York, 1976) pp. 219–226. Zbl. 498.12005.

[Yun, 1976] Yun,D.Y.Y., On Square-free Decomposition Algorithms. Proc. SYMSAC 76 (ACM, New York, 1976) pp. 26–35. Zbl. 498.13006.

[Yun, 1977] Yun,D.Y.Y., On the Equivalence of Polynomial Gcd and Squarefree Factorization Algorithms. Proc. 1977 MACSYMA Users' Conference (NASA Publication CP–2012, National Technical Information Service, Springfield, Virginia) pp. 65–70.