UNIVERSITÀ DEGLI STUDI DI ROMA TRE
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Exploiting SML for Experimenting with Algebraic Algorithms: The Example of p-adic Lifting

WOLFGANG GEHRKE AND CARLA LIMONGELLI

RT-INF-5-96 **1996**

Università degli Studi di Roma Tre
Dipartimento di Informatica
Via della Vasca Navale 84
I - 00146 Roma, Italy
{wgehrke, limongel}@inf.uniroma3.it

both sponsored by Project MURST 40% "Calcolo Algebrico e Simbolico"

ABSTRACT

This paper shows the expressive power of the functional programming language Standard ML (SML) in the context of computer algebra. It is focused on a special application of the p-adic lifting technique, the Hensel algorithm, that is utilized in a symbolic but also numeric context. This experiment demonstrates that SML provides a suitable frame for the implementation of abstract algebraic notions together with the possibility to code related algorithms in a generic way on the corresponding level of abstraction.

1 Introduction

The functional programming language Standard ML (SML) originally evolved as a "meta-language" in the context of logic deductions. Meanwhile it was turned into a general purpose high-level programming language [MTH90, MT91]. It provides imperative features, an exception mechanism, and a powerful parametric module system (examples can be found in [Pau91]).

As the benefits for computer algebra resulting from the application of SML we see the following:

1. The SML notation of algebraic notions comes very close to algebraic specifications.
2. The clear typing of all operations makes the understanding for a newcomer to the field of computer algebra easier.
3. Since SML is widely applied for theorem proving systems this allows the integration of both components: a computational one and a deductive one.

Recently in [San95] SML was used to implement an expressive type system suitable for computer algebra. Furthermore also Extended ML [San89] provides a promising language since it additionally allows logical axioms inside signatures but it currently lacks an implementation.

There are several reasons that carried us to apply SML in the field of computer algebra:

its strong polymorphic type system which enforces a considerable discipline in coding,
its advanced module systems which allows generic programming of algebraic notions on a corresponding level of abstraction,
its formalized semantics which provides a mean to reason about SML programs.

We can also take advantage of further extensions of the language like Concurrent ML [Rep91] or the higher-order module system [Tof94].

In order to verify our claim we have chosen the Hensel algorithm as a suitable candidate. On the one hand this algorithm is defined at a high level of abstraction. On the other hand it is also suitable to perform numeric computations.

In this first experiment we have restricted the implementation to univariate polynomial root finding. The general case of multivariate polynomials has been studied in detail in the literature [Lau83, Yun74]. A special case of this algorithm is the exact representation of algebraic numbers in a p-adic domain.

Our implementation differs from existing computer algebra systems since there the algorithms presented in this paper are mostly built-in. In contrast we make the implementation techniques visible and provide different instantiations for the same abstract notion. Therefore the final algorithm can be easily customized.

From the functional point of view we contribute:

- a set of sufficiently general operations for polynomials in the context of the Hensel algorithm,
- an interesting example of the usefulness of higher-order functors,
- a first step towards a library for specialized computations with algebraic numbers.

Nevertheless the efficiency of such generic algorithms still has to be investigated.

In Sec. 2 we give a brief overview of the Hensel algorithm and in Sec. 3 we summarize main features of SML. The Sec. 4 provides details of the implementation and a discussion of design choices. Furthermore we describe how to exploit the functional style and SML modules. We demonstrate how algebraic notions can be implemented on an appropriate level of abstraction. Examples and tests are shown in Sec. 5. Finally we conclude and suggest future work.

2 Some Background

Given a polynomial equation and a suitable initial approximation $mod\ p^t$ of its solution, with p being prime, lifting algorithms compute a solution $mod\ p^{t+1}$, where p belongs to the domain where the polynomial is defined. They are based on Newton's method for root finding, translated into an appropriate algebraic domain, that is in the most general case a commutative ring. The following theorem states the convergence of the lifting algorithm.

Theorem 1 (Abstract Linear Lifting). *Let I be a finitely generated ideal in a commutative ring R and $f_1, \ldots, f_n \in R[x_1, \ldots, x_r]$, $r \geq 1$, $a_1, \ldots, a_r \in R$ with*

$$f_i(a_1, \ldots, a_r) \equiv 0 \; mod \; I, \quad \text{with } i = 1, \ldots, n.$$

Further let $U = (u_{i,j}), i = 1, \ldots, n,\; j = 1, \ldots, r$, with $u_{i,j} = \frac{\partial f_i}{\partial f_j}(a_1, \ldots, a_r) \in R$ *(U is the Jacobian matrix of $f_1, \ldots, f_n$, evaluated at $a_1, \ldots, a_r$). Assume that U is invertible mod I. Then for each positive integer t, there exist $a_1^{(t)}, \ldots, a_r^{(t)} \in R$, such that*

$$f_i(a_1^{(t)}, \ldots, a_r^{(t)}) \equiv 0 \; mod \; I^t, \quad i = 1, \ldots, n \;\; and \;\; a_j^{(t)} \equiv a_j \; mod \; I, \quad j = 1, \ldots, r.$$

Proof. The proof is given by induction on t. See [Lau83]. □

The approximation methods for p-adic construction are based on the following computational steps:

1. start from an appropriate initial approximation,
2. compute the first order Taylor series expansion,
3. solve the obtained equation,
4. find an update of the solution.

The Hensel algorithm is based on this theorem and its constructive proof. We will consider the restriction to univariate polynomials and algebraic numbers. In particular we will assume $R = \mathbb{Z}[x]$, $I = (p)$ the ideal generated by a prime number, $a_1 = G^1, a_2 = H^1$ both in $R = \mathbb{Z}[x]$ and $\Phi(x, G, H)$ a polynomial function. According to the previous theorem, for any positive integer t, there exists $G^{(t)}, H^{(t)} \in \mathbb{Z}_p^{(t)}[x]$, such that

$$G^{(t)} \equiv G \; mod \; p, \quad \text{and} \quad H^{(t)} \equiv H \; mod \; p.$$

Given $F \in \mathbb{Z}[x]$, $n \in \mathbb{N}$ we want to find the solution of the equation $\Phi(x, G, H) = 0$ where $G, H, \in \mathbb{Z}[x]$ and $x \in \mathbb{Z}$.

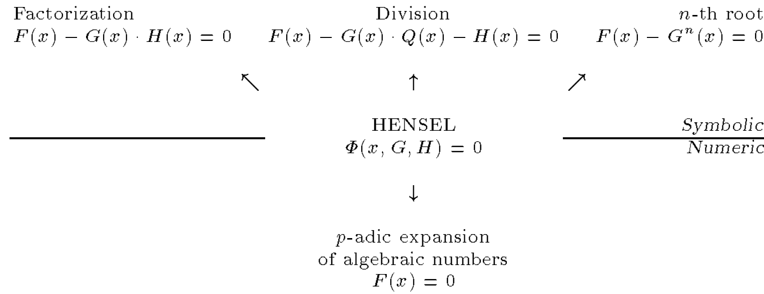

Fig. 1. Applications of Hensel Method

Already at this less general level it is possible to appreciate the intrinsic abstraction of this method that can solve different computing problems. These are either symbolic (factorization, n-th root of a polynomial, polynomial division) or numeric (zero of a polynomial, p-adic expansion of an algebraic number), according to different instantiations of its input parameters as the Fig. 1 shows.

3 Functional Programming with SML

SML is a strict and impure functional programming language, i.e. also functions are first-class objects. It is strict since the evaluation mechanism works based on the rule "call-by-value". It is impure since it contains reference types, exceptions, and an imperative I/O mechanism.

SML is a statically typed language using a sophisticated polymorphic type system where types can be inferred at compile time. The type system is accompanied by a highly advanced module system. In

particular these modules make the language very attractive since they are a powerful tool to structure large programs.

Another important feature of SML is that it has a formal semantics, actually its definition [MTH90, MT91] proceeds in a completely formal style. This is an important point when properties of SML programs have to be proved. A general introduction and examples of SML programs can be found in [Pau91].

The module system as a mean to structure complex programs consists of:

structures as a way to bundle various declarations together,
signatures as the type checking information of structures,
functors as mappings from structures to structures.

The descriptive power of signatures comes close to algebraic specifications. Functors can describe generic constructions of new structures in terms of given ones.

This module discipline allows additionally separate compilation. Finally it should be mentioned that meanwhile the module concept has been generalized to the higher-order case [Tof94]. This means that now also functors can be the input but also output of other functors and they have their own signatures.

For our implementation we have been using the SML/NJ compiler [1]. It comes together with several useful tools, like a make facility, and already supports higher-order functors. For our presentation we took advantage of this generalization.

4 Organization of the Implementation

In the following we comment our implementation. It should be noted that this is not just a description of the code. SML leads the programmer to structure the given problem clearly. A deeper mathematical understanding is necessary to achieve an appropriate implementation.

4.1 Signatures for Algebraic Notions

The initial algebraic notion for our purpose is a *ring* and a *commutative ring* with *unit*. These notions can be represented in a straightforward fashion. Here we present their signatures:

```
signature Ring =
    sig
        type $
        val eq : $ * $ -> bool
        val r2s : $ -> string
        val zero : $
        val neg : $ -> $
        val plus : $ * $ -> $
        val times : $ * $ -> $
    end
```

```
signature UnitCommutativeRing =
    sig
        include Ring
        val one : $
    end
```

In the signature `Ring` we make the equality between elements explicit in form of the function `eq` together with a function `r2s` which allows to print ring elements. A commutative ring with unit has just one more constructor `one` for the type `$`. Properties like commutativity are not represented in this coding.

This last point is controversial. Of course it would also be possible to maintain a set of properties for every structure together with some rules how to compute new properties from given ones. Nevertheless this merely means a hard coding of mathematical theorems.

Another approach to this problem could be the use of Extended ML as demonstrated in [San89]. In Extended ML signatures can contain logical axioms which describe further constraints. Unfortunately there does not exist a system to support Extended ML.

We proceed to the next mathematical notion, an *Euclidean domain*. This notion can be enriched generically by the computation of the *greatest common divisor* and the *extended Euclidean algorithm*. The other two functions are special calls to the extended Euclidean algorithm.

```
signature EuclideanDomain =
    sig
        structure ucr : UnitCommutativeRing
        exception DivMod
        val div : ucr.$ * ucr.$ -> ucr.$
        val mod : ucr.$ * ucr.$ -> ucr.$
```

[1] Copyright 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996 by AT&T Bell Laboratories

```
    end

signature EnrichedEuclideanDomain =
    sig
        include EuclideanDomain
        val gcd : ucr.$ * ucr.$ -> ucr.$
        val eea : ucr.$ * ucr.$ -> ucr.$ * ucr.$ * ucr.$
        exception DiophFail
        val dioph : ucr.$ * ucr.$ * ucr.$ -> ucr.$ * ucr.$
        exception NoEinv
        val einv : ucr.$ * ucr.$ -> ucr.$
    end
```

This representation of an Euclidean domain is slightly different from the literature as in [Lip81]. We do not provide a *degree* function since we do not need it explicitly. `div` and `mod` are implemented as functions and they are not just existential statements.

The integers will be an example of a commutative ring with unit and also of an Euclidean domain. Furthermore we need modular arithmetic. In case of prime numbers with these ingredients we can construct a finite commutative field.

```
signature CommutativeField =
    sig
        structure ucr : UnitCommutativeRing
        exception DivisionByZero
        val inv : ucr.$ -> ucr.$
        type integer
        exception NotPrime
        val char : integer
    end
```

As already done for the Euclidean domain we make use of SML exceptions to code inadmissible computations. Intentionally we have an own type `integer` in order to be flexible in its implementation which here is needed for coding the type for the *characteristic*. The exception `NotPrime` will be used to check a semantical requirement during the creation of a finite field.

Finally we present the signature for polynomials. For some operations we make use of the built-in type `int` for integers since these support other operations like comparison. The substructures `base` and `ucr` describe the rings of coefficients and the univariate polynomials, resp.

```
signature UniPolynomial =
    sig
        structure base : UnitCommutativeRing
        structure ucr : UnitCommutativeRing
        val simplify : ucr.$ -> ucr.$
        val map : (base.$ -> base.$) -> ucr.$ -> ucr.$
        val p2l : ucr.$ -> base.$ list
        (* use dense list of coefficients beginning with the smallest *)
        val l2p : base.$ list -> ucr.$
        val embed : base.$ -> ucr.$
        exception NoProjection
        val project : ucr.$ -> base.$
        val degree : ucr.$ -> int
        val lcf : ucr.$ -> base.$
        val subst : ucr.$ * ucr.$ -> ucr.$
        val shift : ucr.$ * int -> ucr.$
        val derive : ucr.$ -> ucr.$
        val power: ucr.$ * int -> ucr.$
    end

signature PolynomialEuclideanDomain =
    sig
        include UniPolynomial
        structure base´ : CommutativeField
        exception DivMod
        val div : ucr.$ * ucr.$ -> ucr.$
        val mod : ucr.$ * ucr.$ -> ucr.$
    end
```

Several times we made use of substructures with the `UnitCommutativeRing` signature in order to express better sharing constraints in the following code. For the polynomials it was necessary to have conversion functions between the polynomials and its coefficients (`embed` and `project`) and polynomials and a generally available type (`p2l` and `l2p`). Note also the generality of the substitution `subst` which does not just take an element but an entire polynomial.

4.2 Example of a Functor

To illustrate the module concept of SML we present an interesting functor in detail. This is the enrichment of an Euclidean domain by further functions. The enrichment is coded here once and for all. Later it can be applied in different situations for example for integers but also for polynomials over a commutative ring.

```
functor EnrichED(structure ED: EuclideanDomain): EnrichedEuclideanDomain=
    struct
        local open ED.ucr
        in
            fun gcd(a, b) = if eq(b, zero) then a
                            else gcd(b, ED.mod(a, b))
            (* eea(a,b) = (x,y,z) s.t. a*y + b*z = x = gcd(a,b) *)
            fun eea(a, b) =
                let fun aux_loop((a0, a1), (s0, s1), (t0, t1)) =
                    if eq(a1, zero) then (a0, s0, t0)
                    else let val q = ED.div(a0, a1)
                         in
                             aux_loop((a1, plus(a0, neg(times(a1, q)))),
                                      (s1, plus(s0, neg(times(s1, q)))),
                                      (t1, plus(t0, neg(times(t1, q)))))
                         end
                in
                    aux_loop((a, b), (one, zero), (zero, one))
                end
            (* solve the equation f*u + g*v = h if gcd(f,g) divides h *)
            (* returns (u, v)                                          *)
            exception DiophFail
            fun dioph(f, g, h) =
                let val (gcd, s, t) = eea(f, g)
                    val c = ED.div(h, gcd)
                    val b = ED.div(g, gcd)
                    val a = times(c, s)
                    val q = ED.div(a, b)
                    val r = ED.mod(a, b)
                in
                    (r, plus(times(c, t),times(q, ED.div(f, gcd))))
                end handle DivMod => raise DiophFail
            (* computes a^(-1) mod m if this is possible *)
            exception NoEinv
            fun einv(a, m) = let val (gcd, s, t) = eea(m, a)
                             in if eq(gcd, one) then ED.mod(t, m)
                                else if eq(gcd, neg(one))
                                         then ED.mod(neg(t), m)
                                     else raise NoEinv
                             end
        open ED
    end
```

The entire program contains some more auxiliary functors which we will not present in all detail here. These concern the construction of polynomials from rings or fields and the creation of a finite field. The latter is parameterized in an implementation of integers for more flexibility.

Although we do not present the code we give the *functor signatures*. We follow the style of higher-order modules as suggested in [Tof94]. SML/NJ implements this module discipline, too.

```
funsig Enrich (structure ED : EuclideanDomain) =
sig
    include EnrichedEuclideanDomain
    sharing ED.ucr = ucr
end
funsig UniPolys (structure UCR : UnitCommutativeRing)=
sig
    include UniPolynomial
    sharing UCR = base
end
funsig UniPolyd (structure CF : CommutativeField
                 functor UP : UniPolys) =
sig
    include PolynomialEuclideanDomain
    sharing CF.ucr = base and CF = base'
end
funsig ModP (structure IEED : EnrichedEuclideanDomain
             val prime : IEED.ucr.$) =
sig
    include CommutativeField
    sharing type IEED.ucr.$ = ucr.$ and type IEED.ucr.$ = integer
end
```

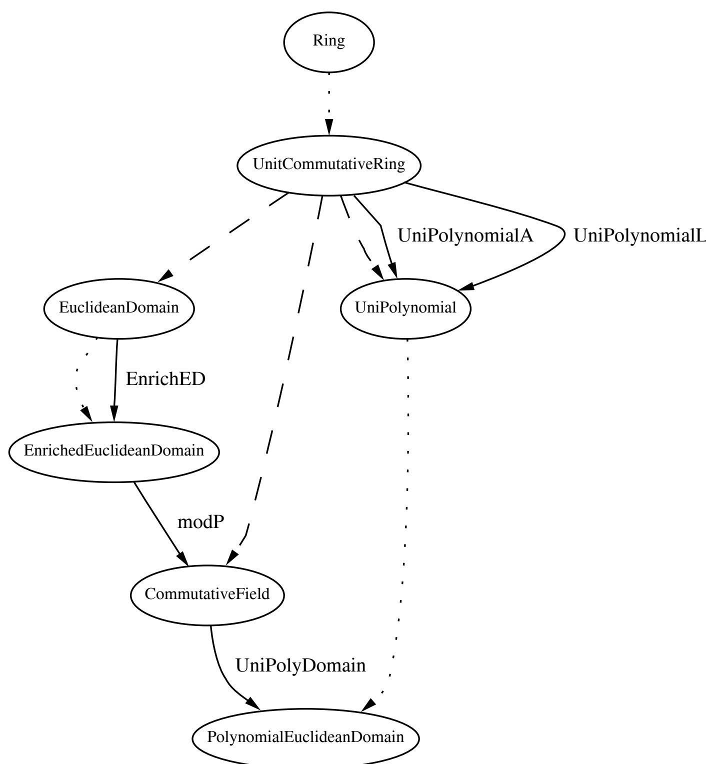

Fig. 2. Main Signatures and Functors

When writing the functor with the signature `UniPolys` we have to decide the way of implementing polynomials. It has been done in two different ways: with lists and with arrays. Since the functor with the signature `UniPolyd` makes use of the previous functor it is turned into a parameter.

This functor `UniPolydomain` is a non-trivial example of an implementation using the higher-order style. On the one hand we want the flexibility of allowing different implementations of polynomials, on the other hand we do not want to code polynomials again. This situation is characterized in that `UniPolydomain` is an extension of a functor implementing polynomials.

The relationship between the signatures and functors can be seen in the Fig. 2. For the `modP` functor there is a further parameter, a prime number. As said before the `UniPolyDomain` functor is also parameterized over a functor with the signature `UniPolys`.

The dotted arrows indicate the inclusion relation between signatures. The dashed arrows mean that there is a substructure of the given signature. The functors are shown by normal arrows.

4.3 The Central Algorithm

The main algorithm is implemented in form of a single functor `Hensel`. Its main input are a representation of the integers and a prime number. The interface of this functor is described in Fig. 3.

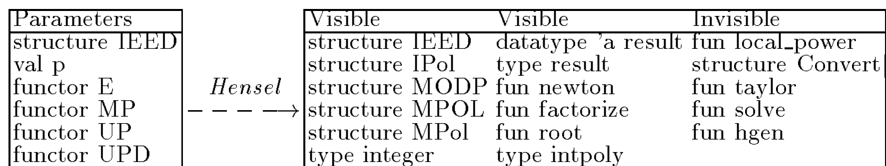

Fig. 3. Hensel Functor

All the other auxiliary structures are created internally with the help of the provided functors:

`IEED` a copy of the parameter,
`IPol` the integer polynomials,
`MODP` the finite commutative field of characteristic p,
`MPOL` the polynomials over this field,
`MPol` these polynomials enriched by further operations.

The type definitions `integer` and `intpoly` make use of these structures. The signature hides a further structure `Convert` which is needed for conversion between integers and modular numbers and for the conversion of the corresponding polynomials.

The main internal functions which are also hidden are:

`taylor` gives the Taylor series expansion,
`solve` computes the solution of the equation obtained by the Taylor series expansion and finds a next approximation,
`hgen` triggers the iteration of this process.

The input of `hgen` is `Phi`, its derivation, the approximations for `x, g, h` and the number of iterations. This coding closely follows the Maple code as in [Lim93].

The functions visible in the interface, `newton`, `factorize`, `divide` and `root` are all specialized calls to `hgen`. Since `Phi` is a function in the variables `x,g,h` the code uses a lambda abstraction. The same applies for the partial derivatives of `Phi`.

This functional representation allows an easy way of implementing all the 4 special cases. Furthermore we can completely avoid multivariate polynomials which would make the code considerably more complex. The provided operations for univariate polynomials suffice to code an arbitrary polynomial function.

Parts of the code for the functor `Hensel` as the heart of the implementation can be found in the Appendix. The use of external functors is made explicit in the parameter list. The other functors are mainly routine implementations of the corresponding notion.

5 Example Instantiations and Tests

We have been instantiating the algorithm by providing different alternative possibilities.

Arithmetic: it is possible to use the integer arithmetic provided by the compiler or instead a package for arbitrary precision arithmetic,
Modular Arithmetic: here it is possible to experiment with different modular representation for numbers. Once p is fixed we can decide to represent the numbers either as $[0, p-1]$ (structure `IntED0`), or as $[-\lfloor \frac{p}{2} \rfloor, \lfloor \frac{p}{2} \rfloor]$ (structure `IntED1`),
Polynomials we can decide for different implementation of polynomials. In our case we decided to implement them in two different ways, by lists and arrays.

The structures `IntEE0` and `IntEE1` result from applying the functor `EnrichED` to `IntED0` and `IntED1`, resp.

We show how the different versions of modular arithmetic are used. We are going to instantiate the functor Hensel and in this case the arithmetic we will use is given by the structure `IntEE0`.

```
structure E5 = Hensel(structure IEED = IntEE0
                      val p = 5
                      functor E = EnrichED
                      functor MP = modP
                      functor UP = UniPolynomialL
                      functor UPD = UniPolyDomain)
```

When we want to modify the number representation, the only necessary change is to alter this parameter. In this case `IntEE1` can be used.

If we want to work with arrays instead of lists we only have to provide the functor `UniPolynomialA` instead of `UniPolynomialL`. This will instantiate the polynomial structures implemented as arrays.

```
structure E5´ = Hensel(structure IEED = IntEE1
                       val p = 5
                       functor E = EnrichED
                       functor MP = modP
                       functor UP = UniPolynomialA
                       functor UPD = UniPolyDomain)
```

Let us show now some examples of the application of the Hensel algorithm itself. We present three main examples, related to the zero of a polynomial, polynomial factorization and a root of a polynomial.

Example 1. We want to find the complex root of the equation $x^2+1=0$.

```
- val x41 = E5.newton(E5.IPol.l2p([1,0,1]), 2, 3);
val x41 = Approx [(2,-,-),(1,-,-),(2,-,-),(1,-,-)] : E5.results
```

Here the structure **E5** is the one described in the previous example. The function **newton** calls the function **hgen** that is the core of the algorithm. The function **newton** shown below, describes the shape of the polynomial function Φ and its derivative by means of a lambda abstraction.

```
fun newton(f, x0, r) =
    hgen((fn (x,g,h)=>IPol.subst(f,IPol.embed(x))),
         ((fn (x,g,h)=>IPol.subst(IPol.derive(f),IPol.embed(x))),
          (fn (x,g,h)=>IPol.ucr.zero),     (* SINCE NOT NEEDED *)
          (fn (x,g,h)=>IPol.ucr.zero)),    (* SINCE NOT NEEDED *)
         x0, IPol.ucr.zero, IPol.ucr.zero, r)
```

Since in this case the partial derivatives with respect to **g** and **h** are not used in the computation, we set them to zero.

```
- val x42 = E5.newton(E5.IPol.l2p([1,0,1]), 3, 3);
val x42 = Approx [(3,-,-),(3,-,-),(2,-,-),(3,-,-)] : E5.results
```

The results are the p-adic expansions of $-i$ and $+i$.

Example 2. For the following factorization we get an exact result.

```
val x5´ = E5´.factorize(E5´.IPol.l2p([~119,309,~163,~22,12,1]),
              (E5´.IPol.l2p([2,0,0,1]),E5´.IPol.l2p([~2,2,1])), 5)
step 0
g = +1 *x^3 +2
h = +1 *x^2 +2*x -2
step 1
g = +2*x -2
h = +2*x -1
step 2
g = -1*x +1
h = +0
```

From the output of the result we can see that

$$\Phi(x,G,H) = x^5+12x^4-22x^3-163x^2+309x-119 =$$

$$\big((x^3+2)\cdot 5^0+(2x-2)\cdot 5^1+(-x+1)\cdot 5^2\big)\cdot\big((x^2+2x-2)\cdot 5^0+(2x-1)\cdot 5^1+0\cdot 5^2\big)$$

Example 3. The example of an exact root is shown below:

```
- val x14 = E5.root((E5.IPol.l2p([49,126,95,18,1]), 2),
                            E5.IPol.l2p([2,4,1]), 3)
exact result
step 0
g = +1 *x^2 +4*x +2
h = +1
step 1
g = +1*x +1
h = +0
```

Again we can verify that:

$$\Phi(x,G,H) = x^4+18x^3+95x^2+126x+49 = \big((x^2+4x+2)\cdot 5^0+(x+1)\cdot 5^1\big)^2$$

Note that in the last two examples the polynomials have been represented by a list of coefficients, starting with the least significant one, where this is the external form of coding polynomials and it does not need to coincide with the internal representation.

6 Conclusions and Future Work

We have demonstrated the useful application of a high-level functional programming language to an abstract algebraic algorithm, the p-adic lifting. The implementation benefits from the functional style as well as from the parametric modularity of the program. SML provides means to describe the procedures on the right level of abstraction.

Furthermore this flexibility allows different instantiations of the overall program. Different implementations of integers can be used (built-in or arbitrary precision) as well as different forms of modular arithmetic (different normal forms). Also the concrete implementation of polynomials can be altered using lists or arrays for example.

Because of its clear typing this implementation can also be used for teaching the Hensel algorithm. Different implementation techniques for polynomials are another possible example. The availability of various instantiations allows a variety of practical tests.

After a detailed performance analysis we plan to extend this kernel to a SML library for specialized computations with algebraic numbers. This performance analysis should not only investigate the use of different implementation methods. Also the influence of generic programming on the performance has to be considered.

As an alternative such an implementation could be contrasted with a realization of computable complex numbers in the style of [Vui87]. Again both approaches have to be analyzed. In form of libraries both are highly desirable for performing exact scientific computations.

References

[Lau83] M. Lauer. Computing by Homomorphic Images. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*. Springer-Verlag, 1983.

[Lim93] C. Limongelli. *The Integration of Symbolic and Numeric Computation by p-adic Construction Methods*. PhD thesis, Università degli Studi di Roma "La Sapienza", 1993.

[Lip81] J.D. Lipson. *Elements of Algebra and Algebraic Computing*. Addison-Wesley Publishing Company, 1981.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pau91] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Rep91] J.H. Reppy. CML: A Higher-Order Concurrent Language. In *SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[San89] D. Sannella. Formal Program Development in Extended ML for the Working Programmer. Technical Report ECS-LFCS-89-102, LFCS, Department of Computer Science, University of Edinburgh, December 1989.

[San95] P.S. Santas. A Type System for Computer Algebra. *Journal of Symbolic Computation*, 19:79–109, 1995.

[Tof94] M. Tofte. Principal Signatures for Higher-Order Program Modules. *Journal of Functional Programming*, 4(3):285–335, July 1994.

[Vui87] J. Vuillemin. Exact Real Computer Arithmetic with Continued Fractions. Technical report, INRIA, Rocquencourt, France, 1987.

[Yun74] D.Y.Y. Yun. *The Hensel Lemma in Algebraic Manipulation*. PhD thesis, Massachusetts Institute of Technology, November 1974.

A Full Code of the Functor Hensel

```
functor Hensel(structure IEED : EnrichedEuclideanDomain
               val p : IEED.ucr.$
               functor E : Enrich
               functor MP : ModP
               functor UP : UniPolys
               functor UPD : UniPolyd) : Hensel =
    struct
        structure IEED = IEED
        structure IPol = UP(structure UCR = IEED.ucr)
        structure MODP = MP(structure IEED = IEED
                            val prime = p)
        structure MPOL = UPD(structure CF = MODP
                             functor UP = UP)
        structure MPol = E(structure ED = MPOL)
        type integer = IEED.ucr.$
        type intpoly = IPol.ucr.$
        (* ... *)
        datatype ´a result =
            (* distinguish the type of the result *)
            Exact of ´a
          | Approx of ´a
        type results = ((integer * intpoly * intpoly) list) result
        fun taylor (x, g, h) Phi Pder pp =
            let val (pdx, pdg, pdh) = Pder
            in
                ((IPol.map (fn xx => IEED.div(xx,pp)) (Phi(x, g, h))),
                 (pdx(x, g, h)), (pdg(x, g, h)), (pdh(x, g, h)))
            end

        fun solve (x, g, h) (phi, dx, dg, dh) =
            if (IPol.ucr.eq(g,IPol.ucr.zero) andalso
                IPol.ucr.eq(h,IPol.ucr.zero)) then
                (Convert.i2m(IEED.ucr.times
                             (IEED.einv(IPol.project(dx), p),
                              IEED.ucr.neg(IPol.project(phi)))),
                 IPol.ucr.zero, IPol.ucr.zero)
            else
                if IPol.ucr.eq(h, IPol.ucr.one) then
                    (x,
                     Convert.mp2ip(MPol.div(MPol.ucr.neg
                                            (Convert.ip2mp(phi)),
                                            Convert.ip2mp(dg))),
                     IPol.ucr.zero)
                else
                    let val (g´, h´) =
                        MPol.dioph(Convert.ip2mp(dg),
                                   Convert.ip2mp(dh),
                                   MPol.ucr.neg(Convert.ip2mp(phi)))
                    in
                        (x, Convert.mp2ip(g´), Convert.mp2ip(h´))
                    end

        (* global entrance point for all calls *)
        fun hgen(Phi, Pder, x0, g0, h0, r) =
            (* Phi as a function in x, g, and h
             F and n are implicitly coded by Phi
             Pder as a triple of functions in x, g, and h
             representing the partial derivatives wrt x, g, h, resp.
             *)
            let fun aux (x, g, h) r Res pp =
                let val exact = IPol.ucr.eq((Phi(x,g,h)), IPol.ucr.zero)
                in
                    if (r = 0) then
                        if exact then Exact(Res)
                        else Approx(Res)
                    else
                        if exact then Exact(Res)
                        else
                            let val phi_exp = taylor (x, g, h) Phi Pder pp
                                val (xs, gs, hs) = solve (x, g, h) phi_exp
                                val nx = IEED.ucr.plus(x,
                                         IEED.ucr.times(xs, pp))
                                val ng = IPol.ucr.plus(g,
                                         IPol.ucr.times(gs,IPol.embed(pp)))
                                val nh = IPol.ucr.plus(h,
                                         IPol.ucr.times(hs,IPol.embed(pp)))
                            in
                                if (IPol.ucr.eq((Phi(nx,ng,nh)),
                                                 IPol.ucr.zero)) then
```

```
                                        Exact(Res @ [(xs, gs, hs)])
                                    else
                                        aux (nx, ng, nh) (r - 1)
                                        (Res @ [(xs, gs, hs)])
                                        (IEED.ucr.times(pp, p))
                                end
                end
            in
                aux (x0, g0, h0) (Integer.abs(r)) [(x0, g0, h0)] p
            end
        (* ... *)
    end
```

This article was processed using the LaTeX macro package with LLNCS style