Towards abstract and executable multivariate polynomials in Isabelle

Florian Haftmann, TU Munich
Andreas Lochbihler, Institute of Information Security, ETH Zurich
Wolfgang Schreiner, RISC, Johannes Kepler University Linz

Abstract. This work in progress report envisions a library for multivariate polynomials developed jointly by experts from computer theorem proving (CTP) and computer algebra (CA). The urgency of verified algorithms has been recognised in the field of CA, but the cultural gap to CTP is considerable; CA users expect high usability and efficiency. This work collects the needs of CA experts and reports on the design of a proof-of-concept prototype in Isabelle/HOL. The CA requirements have not yet been fully settled, and its development is still at an early stage. The authors hope for lively discussions at the Isabelle Workshop.

1 Introduction

The Isabelle distribution [Nip12] and the accompanying Archive of Formal Proof [afp] already contain various formal developments about polynomials; abstract developments include

1. a hierarchy of algebraic modules (locales) (theory *UnivPoly*) featuring an abstract construction of univariate polynomials;
2. a type *'a poly* (theory *Polynomial*) for univariate polynomials over an algebraic structure *'a*; this covers also advanced notions like polynomial division, polynomial derivation and the fundamental theorem of algebra;
3. various application-specific fragments of polynomial theory, including Gröbner bases (theory *Groebner-Basis*), Ferrante and Rackoff's procedure with polynomial parameters (theory *Parametric-Ferrante-Rackoff*), reflected decision procedures (theories *Reflected-Multivariate-Polynomial*, *Commutative-Ring*), a particular executable representation of multivariate polynomials [ST10].

The development 1 is designed for abstract algebraic reasoning; the type 2 is a versatile instrument to cover both abstract properties and concrete computations on polynomials, using type nesting for multivariate polynomials; the examples 3 are very specificly tailored towards their particular application.

What is unsatisfactory is the non-integration of all these formalisations: there are no definitions or lemmas which relate the different approaches, i.e. each of the applications 3 features its own development of polynomial theory without relation to a common base.

The reason why *'a poly* cannot serve as a common base is that it cannot express the concept of a multivariate polynom with an arbitrary number of variables: although e.g. *'a poly poly poly* denotes a polynomial with (at most[1]) three variables, HOL's type system cannot express arbitrary type nesting *'a* $(poly^n)$. Hence we cannot formulate typical algorithms from computer algebra for arbitrary numbers of variables.

Therefore we suggest to develop a type for multivariate polynomials. The type approach is very suitable for formulating algorithms and program generation.

[2] Future work includes inspection of approaches towards polynomials found in systems like Coq.

1.1 Mathematical preliminaries about polynomials

In this section, we sketch the mathematical background relevant for the subsequent discussion. We mostly rely on [Win96], but see also [GCL92, Coh02, Coh03, vzGG13].

The traditional interpretation of a "polynomial" (a Greek-Latin hybrid word meaning "many terms") is that of a symbolic expression composed from variables and constants by addition, subtraction and exponentiation; e.g., $2x^3 - 5x + 7$ denotes a univariate polynomial in three terms. A univariate polynomial $c_n x^n + \ldots + c_1 x^1 + c_0$ can be concisely written with the help of the summation quantifier as $\sum_{i=0}^{n} c_i x^i$. Based on this view we may e.g. conveniently define the product of two polynomials by the equation

$$\left(\sum_{i=0}^{n} a_i x^i\right) \cdot \left(\sum_{j=0}^{m} b_j x^j\right) = \sum_{k=0}^{m+n} \left(\sum_{i\in\mathbb{N}_0, j\in\mathbb{N}_0}^{i+j=k} a_i \cdot b_j\right) \cdot x^k$$

However, while this view is sufficient for the paper and pencil work with polynomials, it has in fact little to do with the actual implementation of a polynomial in computer software, e.g., by an array of coefficients, where the code for polynomial multiplication might look as follows:

```
int[] mult(int[] a, int[] b)
{
  int m = a.length-1; int n = b.length-1;
  int[] c = new int[m+n+1];
  for (int i = 0; i <= m; i++)
    for (int j = 0; j <= n; j++)
      c[i+j] += a[i]*b[j];
  return c;
}
```

On a more fundamental (logical or technological) level, a polynomial is neither a symbolic expression nor an array. These two forms are just two different *representations* of a more fundamental mathematical *concept* of a polynomial. Modern (computer) algebra textbooks thus define polynomials in a more abstract fashion; e.g. in [Win96], we read

[1] since any variable may appear with degree 0 only

[2] FiXme Fatal: Shift to end

> Let R be a ring. A *(univariate) polynomial* over R is a mapping $p : \mathbb{N}_0 \to R, n \mapsto p_n$, such that $p_n = 0$ nearly everywhere, i.e., for all but finitely many values of n.

According to this definition, a univariate polynomial is in essence an infinite sequence of coefficients (the sequence positions denote the corresponding exponents) of which only finitely many are non-zero. The subsequent statement

> If $n_1 < n_2 < \ldots < n_r$ are the nonnegative integers for which p yields a non-zero result, then we usually write $p = p(x) = \sum_{i=1}^{r} p_{n_i} x^{n_i}$.

makes clear that the symbolic representation of a polynomial as a sum is just a convenient notation for paper and pencil work.

To consider polynomials as infinite sequences has substantial advantages, because it simplifies the formal definition of polynomial operations. For instance, the definition of polynomial multiplication given above is just a short form of the definition

$$\text{␣} \cdot \text{␣} : (\mathbb{N}_0 \to R) \times (\mathbb{N}_0 \to R) \to (\mathbb{N}_0 \to R)$$
$$a \cdot b := k \in \mathbb{N}_0 \mapsto \sum_{i \in \mathbb{N}_0, j \in \mathbb{N}_0}^{i+j=k} a_i \cdot b_j$$

Here $\cdot$ is a binary operation on infinite sequences of R values that defines for every position $k \in \mathbb{N}_0$ of the resulting sequence the corresponding coefficient.

The set of polynomials over R with addition and multiplication defined like above form a ring which is denoted by $R[x]$ (actually x is not part of the formal definition but just an indicator that this symbol is used to denote the polynomial which maps exponent 1 to coefficient 1 and every other exponent to coefficient 0).

The abstract concept can be generalized to *multivariate* polynomials in a straightforward way [Win96]:

> An *n-variate polynomial* over the ring R is a mapping $p : \mathbb{N}_0^n \to R, (i_1, \ldots, i_n) \mapsto p_{i_1,\ldots,i_n}$, such that $p_{i_1,\ldots,i_n} = 0$ nearly everywhere. p is written as $\sum p_{i_1,\ldots,i_n} x_1^{i_1} \cdots x_n^{i_n}$ where the formal summation ranges over all tuples $(i_1, \ldots, i_n)$ on which p does not vanish. The set of all n-variate polynomials over R form a ring $R[x_1, \ldots, x_n]$. [...] In fact $R[x_1, \ldots, x_n]$ is isomorphic to $(R[x_1, \ldots, x_{n-1}])[x_n]$.

Thus the formal definition maps every vector of n non-negative exponents (the abstract representation of a *monomial*) to a coefficient. As before, this leads to simple and elegant formal definitions.

An important aspect is the isomorphism stated above. It supports the view on the ring $R[x_1, \ldots, x_n]$ of n-variate polynomials as the ring $(R[x_1, \ldots, x_{n-1}])[x_n]$ of univariate polynomials whose coefficients are from the ring $R[x_1, \ldots, x_{n-1}]$ of $(n-1)$-variate polynomials. Thus, if an operation depends only on the ring properties of the coefficient domain, it naturally generalises to multivariate polynomials. However, this assumption is not true for all operations; e.g. polynomial division is only defined on the ring $K[x]$ where K is a field. However, if R is at least an integral domain (as is the case for $R = K[x_1, \ldots, x_{n-1}]$) then $R[x]$ supports "pseudo-division." This suffices to generalize the Euclidean algorithm for computing greatest common divisors to the multivariate case.

The view of polynomials as mappings from an infinite domain of exponents respectively monomials to the coefficient domain provides mathematical elegance. However, it cannot serve as a representation in the computer. As a first step towards this goal, we have to make the representation finite by only considering a finite subset of the domain that contains all exponents respectively monomials that are mapped to non-zero coefficients.

In a second step, we can represent a univariate polynomial in two ways:

densely As a sequence $[c_0, \dots, c_n]$ of coefficients where n is the degree of the polynomial (i.e., the highest exponent with non-zero coefficient). This representation makes sense (only) if most coefficients in the sequence are not 0.

sparsely As a sequence $[(c_0, e_0), \dots, (c_r, e_r)]$ that contains all pairs (c_i, e_i) of a non-zero coefficient c_i and the corresponding exponent e_i. This representation is preferred if many coefficients are zero.

In the n-variate case, we have two choices (which for $n = 1$ both coincide with the representation of univariate polynomials):

recursively The polynomial is considered as a univariate polynomial whose coefficients are $(n-1)$-variate polynomials, i.e., as an element of $(R[x_1, \dots, x_{n-1}])[x_n]$. The univariate polynomial may be represented densly or sparsely as sketched above (*dense recursive representation* or *sparse recursive representation*).

distributively The polynomial is considered as a mapping from monomials to coefficients, i.e., as an element of $R[x_1, \dots, x_n]$. Theoretically, we may choose a *dense distributive representation* by fixing a total order of the monomials and listing all exponents in the respective order until the last exponent vector with a non-zero coefficient (this assumes that any monomial with non-zero coefficients has only finitely many predecessors in the chosen order). However, due to the large number of monomials, a more practical representation is the *sparse distributive representation* which lists all pairs of exponent vectors and non-zero coefficients (if we want to make this representation unique, also here a total monomial order is required).

While many operations on multivariate polynomials are more naturally implemented on a recursive representation, there are also exceptions. In particular, Buchberger's *Gröbner Bases* algorithm is realistically implemented only with a distributive representations, because it processes the individual terms of a polynomial in a total order (that has to satisfy certain properties); thus the polynomial should be represented in a (sparse) distributive way by listing the monomials in the considered order.

1.2 Requirements and goals for a polynomial package

The core goal for the envisioned Isabelle package for multivariate polynomials is to have a single computer-supported framework in which the working mathematician can both of the following.

1. Develop mathematical theories in a style that is close to (modern) mathematical practice, but on the basis of a sound logical and technological framework where e.g., definitions and theorems are mechanically type checked and proofs are developed with computer support and thus mechanically verified.
2. Describe algorithms conveniently based on the developed mathematical notions such that these algorithms (i) can be executed with reasonable efficiency (in the sense of a rapid prototype, not of production quality code) and (ii) can be specified formally and verified with computer assistance.

We approach these goals by providing

an abstract type of multivariate polynomials which is based on their modern mathematical view (as mappings from an infinite domain of monomials to coefficients, §2.1) such that on this type

- a) the fundamental operations can be elegantly defined,
- b) corresponding theorems can be conveniently formulated and proved,
- c) algorithms (functional programs) can be formulated, and
- d) these algorithms can be specified and verified to satisfy their specification;

multiple representation types of multivariate polynomials together with corresponding implementations of the fundamental operations (§2.2) such that

- a) the representation types can be proved to be refinements of the abstract types such that variables of the abstract type can be instantiated with values of any representation type and consequently all the theorems formulated on the abstract types also hold for the representation types, and
- b) from the representation types and the corresponding operations executable code can be generated, and thus
- c) the algorithms formulated on the abstract types become executable by plugging in values of the executable representation types.

Clearly, algorithms should be formulated and verified on an *abstract* type rather than a concrete representation. When instantiated with polynomials in the "wrong" representation, the generated code might be inefficient but nevertheless operable. This requires to carry to the abstract type also several notions that originally stem from particular representation types (e.g., the coefficient of a "main variable", which stems from the recursive representation, or the "leading term" with respect to a monomial ordering, which stems from the distributed representation). Nevertheless it shall also be possible to explicitly coerce polynomials from any (abstract or representation) type to a particular representation type such that one can formulate individual algorithms that (from the point of complexity) crucially depend on a particular representation (respectively formulate composed algorithms where particular parts depend on particular representations).

Apart from these high-level strategic goals, there are also some low-level technical requirements that have to be considered in the subsequent design decisions. In particular:

1. In a system with a static type system (such as Isabelle), the polynomial ring $R[x_1, \ldots, x_n]$ must be mapped to an adequate type that is expressible in the type system. Clearly, such a type $P(R)$ should explicitly depend on (a static type for the coefficient ring) R. Furthermore, as discussed above, in the modern view the variables $x_1, \ldots, x_n$ do not play a role in the representation and can be ignored. However, the number of variables n has to be modelled somehow. We see the following options.

   Dependent type A dependent polynomial type $P(n, R)$ encodes the number of variables. However, Isabelle does not support such types.

   Type nesting Given a type $P(R)$ of univariate polynomials in R, n-variate polynomials have the type $P(P(\ldots(R))$. This relies on $P(\ldots(R))$ itself being a ring. Then the type system prevents e.g. the illegal addition of a univariate polynomial and a bi-variate polynomial, as their types $P(R)$ and $P(P(R))$ differ. As discussed in §1, functions for polynomials with an arbitrary number of types cannot be expressed in this approach.

   Implicit The number of variables does not show up in the type, i.e., $P(R)$ denote the type of multivariate polynomials in R with an arbitrary number of variables. Instead, a function $\mathit{vars} :: P(R) \Rightarrow \mathbb{N}$ returns the number of variables of a polynomial.

   While this seems the most pragmatic solution, it has to address potential consistency issues with respect to the number of variables, both inside a single polynomial (how is this number modeled and does the representation conform to it) and across polynomials (how to deal with situations when an operation is applied to polynomials with different numbers of variables).

2. Since in some parts of polynomial theory a monomial order is relevant (in particular the notion of the “leading monomial” with respect to such an order in Gröbner bases theory), one has to consider whether and how to also model this order on the level of the abstract polynomial type (in addition to the modeling of that order on the distributive representation type). Since, e.g., the Gröbner bases algorithm is only efficiently executable with respect to the monomial order in which the monomials are actually listed in the distributed representation type, this order must also appropriately modeled in the abstract type and cannot be just passed as an independent argument to the algorithm.

Above goals and strategies have been deliberately formulated in an essentially system-agnostic style; in the next section, we describe how to approach them in Isabelle.

2 Design of the prototype

Having stated the requirements and goals, we now present our design for multivariate polynomials in Isabelle/HOL. The abstract type of polynomials (§2.1) captures the modern mathematical view on polynomials. Several representation types (§2.2) provide efficient implementations; refinement in the code generator connects them to the abstract type.

2.1 A type of abstract polynomials

We present the construction of a type $'a\ \mathit{mpoly}$ for multivariate polynomials. Here $'a$ is (the type of) the carrier over which the multivariate polynomials are constructed. Variables are represented by natural numbers:

- Concrete names are an aspect of presentation rather than representation, e.g. the polynomials $x^2 + \mathit{2}$ are $y^2 + \mathit{2}$ considered equivalent.
- Specific algorithms have a notion of variable order, and this is most naturally expressed using natural numbers.

Essential for our specification of multivariate polynomials is an auxiliary intermediate type $'b \Rightarrow_0 {'a}$ isomorphic[3] to $'b \Rightarrow {'a}$ such that each function of $'b \Rightarrow_0 {'a}$ is "almost everywhere zero", ie. $\forall f :: {'b} \Rightarrow_p {'a}.\ \mathit{finite}\ \{a.\ f\ a \neq \mathit{0}\}$[4]. This notion of "almost everywhere zero" is omnipresent: a power product is an almost everywhere zero mapping from variable identifiers to exponents ($\mathit{nat} \Rightarrow_0 \mathit{nat}$), and a (multivariate) polynomial in turn is isomorphic to an almost everywhere zero mapping from power products to coefficients ($(\mathit{nat} \Rightarrow_0 \mathit{nat}) \Rightarrow_0 {'a}$).

2.1.1 Addition

We obtain the additive structure of polynomials $(\mathit{nat} \Rightarrow_0 \mathit{nat}) \Rightarrow_0 {'a}$ by instantiating $'b \Rightarrow_0 {'a}$ pointwise:

$$\mathit{0} = (\lambda a.\ \mathit{0})$$

$$(f + g)\ a = f\ a + g\ a$$

$$(-\ f)\ a = -\ f\ a$$

$$(f - g)\ a = f\ a - g\ a$$

2.1.2 Multiplication

We define multiplication on $'b \Rightarrow_0 {'a}$ as a fold product, delegating to addition on $'b$ and multiplication on $'a$:

$$\mathit{1} = (\lambda a.\ \mathit{1})$$

$$(f * g)\ a = (\textstyle\sum (b,\ c) \in \{(b,\ c).\ b + c = a\}.\ f\ a * g\ b) \quad (\star)$$

[3] notationally we treat $'b \Rightarrow_0 {'a}$ as a subtype of $'b \Rightarrow {'a}$, using immediate application syntax etc.

[4] 0 is overloaded in Isabelle, which imposes a type class constraint *zero* on the result type $'a$, which is omitted here for succintness; similar for class constraint imposed by generic algebraic operations like $+$, $*$ etc.

Here the definition of addition on $'b \Rightarrow_0 'a$ plays a role also: mapping the fold product ($\star$) to polynomials ($(nat \Rightarrow_0 nat) \Rightarrow_0 'a$), the referenced addition on $nat \Rightarrow_0 nat$ turns into addition of exponents of variables, and the referenced multiplication on $'a$ to multiplication of coefficients.

Type $'b \Rightarrow_0 'a$ is also used as base for the more specific type *'a nat-mapping* used for concrete representations[5].

2.2 Representations for efficient computation

The abstract type *'a mpoly* captures the mathematical notion of a multivariate polynomial, but there is too little structure to efficiently implement operations on polynomials. In this section, we add such structure in several refinement steps. The design goals are the following.

Abstraction Abstract polynomials (type *'a mpoly*) are as abstract as possible. Otherwise, details of the representation would clutter the proofs.

Ease of use Algorithms formalised on *'a mpoly* use the efficient implementations with little or no intervention of the user.

User control The user of the library can control the choice of representations and convert one into another.

Refinement can be done inside Isabelle's logic [Lam13] or in the code generator [HKKN13]. We pick the latter option, because it has already been used successfully in the context of container data structures [Loc13].

The code generator provides two kinds of refinement [HKKN13, HN10]. First, *program refinement* separates the definition and implementation of functions. Any (executable) equational theorem suffices for code generation, it need not be the definition. Second, *data refinement* come in two forms. The user may declare pseudo-constructors for any type, in terms of which this type will be implemented in the code. Neither need the new constructors be injective and pairwise disjoint, nor exhaust the type in the logic; their signature merely has to meet the requirements for a constructor. Functions on such a refined type can pattern-match on these new pseudo-constructors in their code equations. Alternatively, types can be made abstract by declaring a representation function (rather than constructors) whose codomain determines the implementation type. Since values of abstract types can be manipulated only by going through the representation function, its range encodes an invariant on the implementation type such as a sortedness of lists. Both forms can be combined by introducing an intermediate type constructor. Note that refinement affects only code generation, but not the logical properties of the refined type. Consequently, one cannot exploit the type's new structure inside the logic.

Several refinement steps are needed to go from *'a mpoly* to implemented algorithms; Fig. 1 shows our current design. The representation types *'a poly-rec* and *'a poly-distr* model the recursive and distributive representation of a polynomial.

[5] FiXme Fatal: not yet known how notation for nat_mapping will look like; place reference here

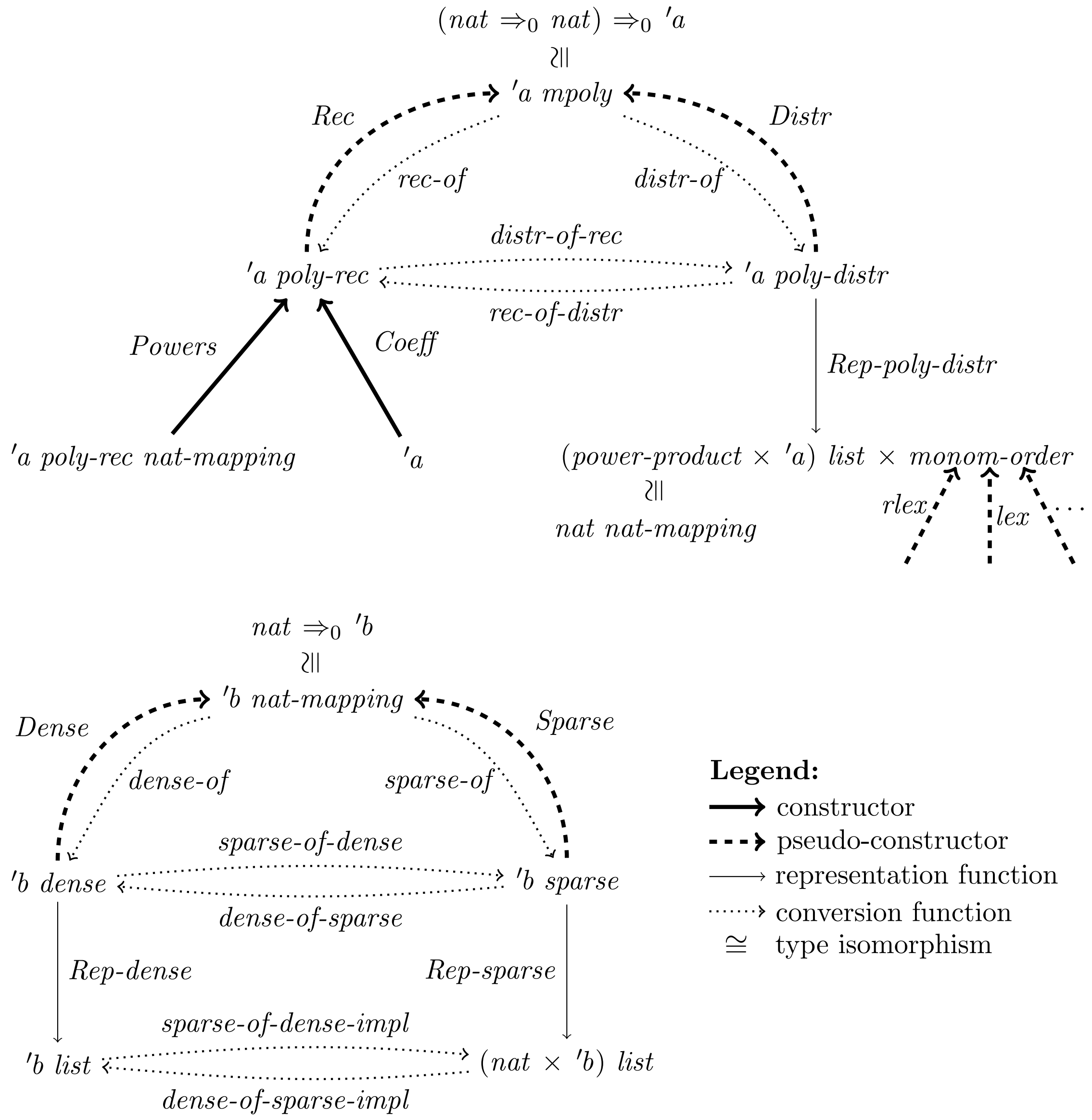

Figure 1: Overview of the different representations of polynomials and their implementation types

The recursive representation is modelled as a datatype with two constructors *Powers* :: *'a poly-rec nat-mapping* ⇒ *'a poly-rec* and *Coeff* :: *'a* ⇒ *'a poly-rec*. The constructor *Powers* models the recursive view of a univariate polynomial (expressed as ... *nat-mapping* which is isomorphic to *nat* $\Rightarrow_0$ *'b*) with recursive polynomials as coefficients.[6] Recursion terminates at a constant, i.e., a polynomial without variables, which *Coeff* embeds into the type of polynomials.

The distributive representation consists of a finite map from power products to coefficients (type *power-product* $\Rightarrow_0$ *'a*) and a monomial ordering (type *monom-order*). This type is implemented as an associative list (*power-product* × *'a*) *list* and a monomial ordering with the invariant that the power products are sorted according to the monomial ordering and no power product is mapped to *0*.[7] These types explicitly include the monomial order such that (i) users can choose a monomial ordering, and (ii) algorithms like Buchberger's can exploit the chosen ordering.

Two abstraction functions *Rec* :: *'a poly-rec* ⇒ *'a mpoly* and *Distr* :: *'a poly-distr* ⇒ *'a mpoly* abstract the representation details. They link the representations with the abstract type and are declared as pseudo-constructors (we call them pseudo-constructors because they are not disjoint). Operations on *'a mpoly* select the algorithm on the representation by pattern-matching. For example, the code equations for addition

$$\begin{array}{lll} \mathit{Rec}\ p & +\ \mathit{Rec}\ q & = \mathit{Rec}\ (\mathit{plus\text{-}poly\text{-}rec}\ p\ q) \\ \mathit{Distr}\ p & +\ \mathit{Distr}\ q & = \mathit{Distr}\ (\mathit{plus\text{-}poly\text{-}distr}\ p\ q) \end{array} \tag{1}$$

call the implementations *plus-poly-rec* and *plus-poly-distr* on the representations. Such code equations ensure that algorithms formulated in terms of executable operations on *'a mpoly* immediately use the representations in the generated code—without any user configuration.

The abstraction functions have counterparts *rec-of* and *distr-of* that are used to convert between representations (*distr-of* is parametrised by a monomial order). They are right-inverses of the abstraction functions.

$$\mathit{Rec}\ (\mathit{rec\text{-}of}\ p) = p \qquad \text{and} \qquad \mathit{Distr}\ (\mathit{distr\text{-}of}\ mo\ p) = p$$

Unfortunately, the recursive representation is not unique, i.e., *Rec* is not injective. For example, *Rec* (*Powers 0*)= *Rec* (*Coeff 0*). Consequently, the conversion of an abstract polynomial represented recursively to the recursive representation must normalise the representation, e.g. by replacing all *Powers* subtrees of constant polynomials with *Coeff*. Hence, the following equations implement *rec-of* where *rec-of-distr* implements the

[6] In principle, the sub-polynomials of a recursive polynomial could be represented distributively, too, i.e., *Powers* had the type *'a mpoly nat-mapping* ⇒ *'a poly-rec*. We decided not to support this, because it would complicate recursive algorithms that are not just straightforward generalisations from the univariate case.

[7] More efficient data structures like binary search trees would also be possible.

conversion from distributive to recursive functions.

$$\begin{aligned} \mathit{rec\text{-}of}\ (\mathit{Rec}\ p) &= \mathit{normalise\text{-}rec}\ p \\ \mathit{rec\text{-}of}\ (\mathit{Distr}\ q) &= \mathit{rec\text{-}of\text{-}distr}\ q \end{aligned}$$

In contrast, the distributive representation is isomorphic to the abstract polynomials for a fixed monomial order. Consequently, converting a distributive polynomial to the distributive representation is a no-op provided that the monomial order matches.

$$\begin{aligned} \mathit{distr\text{-}of}\ \mathit{mo}\ (\mathit{Distr}\ p) &= (\textsf{if}\ \mathit{mo} = \mathit{monom\text{-}order\text{-}distr}\ p\ \textsf{then}\ p\ \textsf{else}\ \mathit{convert\text{-}mo}\ \mathit{mo}\ p) \\ \mathit{distr\text{-}of}\ \mathit{mo}\ (\mathit{Rec}\ q) &= \mathit{distr\text{-}of\text{-}rec}\ q \end{aligned}$$

The code equation tests equality of two the monomial orders. However, the type *monom-order* is defined as a set of functions, namely of all comparison operators that satisfy the monomial order constraints. Hence, equality is undecidable. We nevertheless can implement equality on those few monomial orders that are relevant in practice. To that end, we name these (by defining constants such as *rlex* and *lex*) and declare these names as pseudo-constructors of *monom-order* (they are pseudo-constructors because they do not exhaust the type). Thus, equality becomes pattern-matching on names in the code. This approach achieves that new monomial orders can always be added later. The alternative of defining the datatype of the names in HOL would not.

Yet, these conversion function *rec-of* and *distr-of* break the abstraction of *'a mpoly*. Therefore, the user should not have to call them directly. Instead, we provide cast operations *rec-cast* :: *'a mpoly* ⇒ *'a mpoly* and *distr-cast* :: *monom-order* ⇒ *'a mpoly* ⇒ *'a mpoly*, which are logically the identity on *'a mpoly*. Their code equations call the conversion function and wrap the result with the corresponding abstraction function.

2.3 Dense and sparse representations

In the previous section, we have discussed how recursive and distributive representations for polynomials fit under one hood of abstract polynomials. It turns out that the same ideas also work for handling dense and sparse representations uniformly (see the lower part in Fig. 1). Here, we restrict ourselves to AEZ maps with natural numbers as keys. In the distributive representation, the keys are the variables of the power product; and in the recursive one, they are the exponents of the univariate argument to *Powers*. Since data refinement requires a type constructor, we introduce the type *'b nat-mapping*, which is isomorphic to $\mathit{nat} \Rightarrow_0 {'b}$ (and to *'b poly* of univariate polynomials). Its non-disjoint pseudo-constructors are the abstraction functions from the representation types. The dense representation consists of a plain list of values, on which we impose the invariant that the last element is not *0*. As sparse representation, we use an associative list whose keys appear in ascending order and whose values do not contain *0*. The invariants ensure that the representation types are isomorphic to the abstract type. Hence, the conversion functions become no-ops when the argument already has the right representation. Thanks to *'b nat-mapping*, we can implement dense and sparse representations uniformly for recursive polynomials and for power products.

3 Problems and open questions

The current design of the prototype is not completely satisfactory. There are unsolved issues with respect to the implementation in Isabelle (§3.1 and §3.2), and some of the requirements from CA are not addressed (§3.3). By presenting them below, we hope to generate input from the community.

3.1 The ubiquitous type class *zero*

In the auxiliary type $'b \Rightarrow_0 {'a}$, the *zero* type class determines the value that the function must take almost everywhere. This sort constraint propagates through most types including auxiliary data structures and even their implementation types due to the invariants. This causes two problems.

First, the current design prevents re-use in two ways. On the one hand, it is hard to re-use our data structures in other contexts, although most of them implement variations of finite maps. Ordinary finite maps (a subtype of $'a \Rightarrow {'b}\ \mathit{option}$) uniformly store all values of the codomain and distinguish lookup failure with *None*. In that sense, our types conceptually remove *0* from the values. Hence, if we define $0 = \mathit{None}$, the type $'a \Rightarrow_0 {'b}\ \mathit{option}$ acts like a finite map, but the signatures remain unnatural for other users.

On the other hand, it is also hard to re-use existing data structures (like red-black trees) in our prototype, because the representation types rely on the invariant that *0* is not stored as a value. Possibly, this could be improved by splitting this invariant off from the data structure invariant, i.e., by introducing another intermediate type. It is not clear how this affects performance, as implementations cannot exploit the combined invariants any more. Independent of the sort constraint *zero*, supporting multiple monomial orders complicates things, too. We might be able to reuse the implementation of efficient data structures, but not the non-negligible wrapping-up of invariants which takes the order from the standard type classes.

Second, the sort constraint also prevents the integration with Isabelle packages such as lifting and BNF. On $'a \Rightarrow_0 {'b}$, there is no map function *map* that commutes with function composition, i.e., that satisfies

$$\mathit{map}\ f \circ \mathit{map}\ g = \mathit{map}\ (f \circ g)$$

for all $f :: {'c} \Rightarrow {'d}$ and $g :: {'b} \Rightarrow {'c}$. The problem is that $(f \circ g)\ 0 = 0$ does not imply $f\ 0 = 0$ and $g\ 0 = 0$. Hence, the intermediate $\mathit{map}\ g\ p$ might not be *0* almost everywhere.

Consequently, $'a \Rightarrow_0 {'b}$ and similarly $'b$ *nat-mapping* are not bounded natural functors. Thus, they cannot be registered with the new datatype package [BHL$^+$14]. So, datatype definitions cannot recurse through these types. This is why we were not able to define the recursive representation with a package. Rather, we constructed it manually by carving out the relevant subset from the following datatype and replacing the list with *nat-mapping*, which essentially undoes the dense refinement from Fig. 1.

datatype *'a poly-rec-raw = Coeff-raw 'a | Powers-raw ('a poly-rec-raw list)*

Similarly, there is no primitive recursor on $'a \Rightarrow_0 'b$, either, and support for recursive function definitions must installed manually.

For the same reason, we cannot register a quotient theorem with the lifting package [HK13], either (we can prove one when the quotient relation respects *0*, but the package rejects such additional assumptions). Thus, transfer and lifting cannot work on types that are nested in $'a \Rightarrow_0 'b$ or *'b nat-mapping*. For example, they cannot help us with constructing *'a poly-rec* from *'a poly-rec-raw*.

3.2 Strategies for selection of representations

Early on, we decided that representations should never be converted automatically into one another. This gives the user full control over the chosen representations and avoid surprises in efficiency and run-time. However, we are no longer sure that this is the best option. For example, binary operations like addition are implemented only when both operands have the same representation (1). Consequently, if a distributive polynomial and a recursive one are to be added, the code raises an exception at run time. Such run-time errors can be hard to debug, and Isabelle provides no support as this happens outside the logic. To prevent such exceptions, the code equations must implement all cases. However, in the missing cases, it is not clear how to choose the representation for the result of the computation. This problem also arises when polynomials are required as input. For example, it is tempting to write *2* to denote the constant polynomial with coefficient *2*. Yet, the code generator does not accept this, because the user has not specified whether *2* should be represented recursively or distributively. Therefore, the input syntax is currently cumbersome to write, and so is reading the output syntax.

Moreover, it turns out that the code equation should be able to query the representation details. Consider, e.g., the addition *plus-poly-rec* on recursive polynomials in Fig. 2. When a polynomial p in n variables is added to a polynomial q in $n + m$ variables, p must be extended to $n + m$ variables. The function *plus-poly-rec* silently does so when a coefficient *Coeff c* is added to a univariate polynomial *Powers q* via the identity $c = cx^0$. That is, a coefficient becomes a univariate polynomial, so the representation (dense or sparse) must be chosen. Clearly, it should be the same as q's, but q's is not known in the logic. Therefore, we define single-element types with pseudo-constructors to make representation information accessible in code equations; the overloaded function *implT* extracts this information from abstract values. The functions to construct values of abstract types (like *single* for *nat-mapping*) pattern-match on the pseudo-constructors and thus pick the specified representation. Unfortunately, this technique addresses just this special case—a general solution is still missing.

3.3 Remaining requirements from computer algebra

The Isabelle team quickly developed the prototype sketched in §2. The CA experts found the speed challenging; they still find it hard to relate the goals in §1.2 to the details of the design and the prototype. Below, we list the requirements that have not yet been addressed.

$$\begin{aligned}
&\textit{plus-poly-rec}\ (\textit{Coeff}\ c)\quad\ \ (\textit{Coeff}\ d)\quad\ \ = \textit{Coeff}\ \ (c + d)\\
&\textit{plus-poly-rec}\ (\textit{Powers}\ p)\ (\textit{Powers}\ q) = \textit{Powers}\ (\textit{zip-with}\ (\lambda\_.\ \textit{plus-poly-rec})\ p\ q)\\
&\textit{plus-poly-rec}\ (\textit{Coeff}\ c)\quad\ \ (\textit{Powers}\ q) =\\
&\qquad\qquad \textit{Powers}\ (\textit{zip-with}\ (\lambda\_.\ \textit{plus-poly-rec})\ (\textit{single}\ (\textit{implT}\ q)\ \textit{0}\ (\textit{Coeff}\ c))\ q)\\
&\textit{plus-poly-rec}\ (\textit{Powers}\ p)\ (\textit{Coeff}\ d)\quad\ =\\
&\qquad\qquad \textit{Powers}\ (\textit{zip-with}\ (\lambda\_.\ \textit{plus-poly-rec})\ p\ (\textit{single}\ (\textit{implT}\ p)\ \textit{0}\ (\textit{Coeff}\ d)))
\end{aligned}$$

Figure 2: Addition operation on recursive polynomials

Exploit representational notions at the abstract level The distributive and recursive representations each have specific notions such as leading term and main variable (see p. 5). In the current design, these notions are not part of *'a mpoly* and thus cannot be properties of a polynomial. It is unclear how algorithms on abstract polynomials can exploit such notions, but this is definitely needed. Explicit parameters like for *distr-of* might be a solution.

Convenient input and output formats Prototyping benefits not only from short notations, as mentioned in §3.2; comprehensible visual representation of input and output is equally important. Parsing and pretty-printing features are currently missing, but first, a convenient syntax must be found.

Different kinds of divisions Division on polynomials is unique only in the univariate case over a field. In other cases, the Euclidean algorithm can be formulated by using a generalisation called pseudo-division. Since *'a mpoly* does not distinguish between uni- and multivariate polynomials, we can only have pseudo-division on the abstract type. However, computer algebraists may want to use proper division when they know that the polynomial is univariate. With a good solution, both use cases can be handled uniformly. Maybe, a new type class *pseudo-ring* can capture the nice properties of division and remainder in either case.

4 Current state of affairs

This paper takes a snapshot of an ongoing development for a polynomial library. Both the design and the requirements are not set in stone, and we will try to align them further. The open issues may serve as a starting point to explore new ideas, and we are interested in new ideas and pointers by people in CTP or CA. This also includes inspection of approaches towards polynomials found in other theorem provers.

Finally, a note on the cooperation between Isabelle and CA experts in this project. On the one hand, we have found that CA experts consider the verification of CA software as urgent. Being mathematicians, they are used to proving, but they see that the implementations of their algorithms contradict to their professional honour. On the other hand, they would use verified CA software in practice only if they provide a level of usability and availability comparable to standard tools like SAGE or Singular—at least in the domain of their personal expertise.

Academic CA and CA software development have become large-scale ventures with high impact on science, technology and engineering. As the systems become increasingly complex,

we need a coherent conceptual framework and a gap-less chain of tools
from proving properties of algorithms to generating efficient code.

We have found that Isabelle already provides both a logical base for a framework covering theories and algorithms in CA, and core technologies for the tool chain, in particular automated code generation.

References

[afp] Archive of formal proofs.

[BHL$^+$14] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP 2014)*, LNCS. Springer, 2014.

[Coh02] Joel S. Cohen. *Computer Algebra and Symbolic Computation: Elementary Algorithms*. A. K. Peters, Natick, Massachusetts, 2002.

[Coh03] Joel S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. A. K. Peters, Natick, Massachusetts, 2003.

[GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Springer, 1992.

[HK13] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs (CPP 2013)*, volume 8307 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2013.

[HKKN13] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 100–115, 2013.

[HN10] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer Berlin / Heidelberg, 2010.

[Lam13] Peter Lammich. Automatic data refinement. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2013.

[Loc13] Andreas Lochbihler. Light-weight containers for Isabelle: efficient, extensible, nestable. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*, pages 116–132. Springer, 2013.

[Nip12] Tobias Nipkow. Programming and proving in Isabelle/HOL. contained in the Isabelle distribution, May 22 2012.

[ST10] Christian Sternagel and René Thiemann. Executable multivariate polynomials. *Archive of Formal Proofs*, August 2010. http://afp.sf.net/entries/Polynomials.shtml, Formal proof development.

[vzGG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, 3rd edition, 2013.

[Win96] Franz Winkler. *Polynomial Algorithms in Computer Algebra.* Texts and Monographs in Symbolic Computation. Springer-Verlag, Wien, New York, 1996.