Functional Pearl: Folding Polynomials of Polynomials

Chen-Mou Cheng[1], Ruey-Lin Hsu[2], and Shin-Cheng Mu[3]

[1] National Taiwan University, Taiwan
[2] National Central University, Taiwan
[3] Academia Sinica, Taiwan

Abstract. Polynomials are a central concept to many branches in mathematics and computer science. In particular, manipulation of polynomial expressions can be used to model a wide variety of computation. In this paper, we consider a simple recursive construction of multivariate polynomials over a base ring such as the integers or a (finite) field. We show that this construction allows inductive implementation of polynomial operations such as arithmetic, evaluation, substitution, etc. Furthermore, we can transform a polynomial expression into in a sequence of arithmetic expressions in the base ring and prove the correctness of this transformation in Agda. Combined with our recursive construction, this allows for compiling polynomial expressions over a tower of extension fields into scalar expressions over the ground field, for example. Such a technique is not only interesting in its own right but also finds plentiful application in research areas such as cryptography.

1 Introduction

A *univariate polynomial* over a base ring R is a finite sum of the form

$$a_n X^n + a_{n-1} X^{n-1} + \cdots + a_0,$$

where $a_i \in R$ are the coefficients, and X is called an *indeterminate*. The set of univariate polynomials over R forms a ring, denoted as $R[X]$. We can allow two or more indeterminates $X_1, X_2, \ldots, X_m$ and thus arrive at a *multivariate polynomial*, a finite sum of the form

$$\sum_i a_i X_1^{e_1^{(i)}} X_2^{e_2^{(i)}} \cdots X_m^{e_m^{(i)}},$$

where $a_i \in R$ are the coefficients, and the nonnegative integers $e_j^{(i)}$ are the exponents. The set of m-variate polynomials over R, denoted as $R[X_1, X_2, \ldots, X_m]$, also forms a ring, referred to as a *ring of polynomials*.

Polynomials are a central concept to many branches in mathematics and computer science. In particular, manipulation of polynomial expressions can be used to model a wide variety of computation. For example, every element of an algebraic extension field F over a base field K can be identified as a polynomial over K, e.g., cf. Theorem 1.6, Chapter 5 of the (standard) textbook by Hungerford [6]. Addition in F is simply polynomial addition over K, whereas multiplication in F is polynomial multiplication modulo the defining polynomial of F over K. Let us use the familiar case of the complex numbers over the real numbers as a concrete example. The complex numbers can be obtained by adjoining to the real numbers a root i of the polynomial $X^2 + 1$. In this case, every complex number can be identified as a polynomial $a + bi$ for a, b real. The addition of $a_1 + b_1 i$ and $a_2 + b_2 i$ is simply $(a_1 + a_2) + (b_1 + b_2)i$, whereas the multiplication, $(a_1 + b_1 i)(a_2 + b_2 i) \bmod i^2 + 1 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i$.

In addition to arithmetic in an algebraic extension field, manipulation of polynomial expressions also finds rich application in cryptography in particular. A wide variety of cryptosystems can be implemented using polynomial expressions over a finite field or $\mathbb{Z}/n\mathbb{Z}$, the ring of integers modulo n. In elliptic curve cryptography [8], for example, we use the group structure of certain elliptic curves over finite fields to do cryptography, and the group laws are often given in polynomial expressions over finite fields. Another example is certain classes of post-quantum cryptosystems, i.e., those expected to survive large quantum computers' attack. Among the most promising candidates, we have multivariate cryptosystems [3] and several particularly efficient lattice-based cryptosystems [7, 4], for which encryption and decryption operations can be carried out using polynomial expressions over finite fields or $\mathbb{Z}/n\mathbb{Z}$.

This pearl is initially motivated by our research in cryptography, where we often have to deal with multivariate polynomials over various base rings, as exemplified above. We also need to transform a polynomial expression into a sequence of arithmetic expressions over its base ring. This is useful for, e.g., software implementation of cryptosystems on microprocessors with no native hardware support for arithmetic operations with polynomials or integers of cryptographic sizes, which typically range from a few hundreds to a few thousands of bits. Again using multiplication of two complex numbers as an example, we would need a sequence of real arithmetic expressions for implementing $z = z_r + iz_i = (x_r + ix_i) \times (y_r + iy_i) = xy$:

$$
\begin{aligned}
t_1 &\leftarrow x_r \times y_r; \\
t_2 &\leftarrow x_i \times y_i; \\
t_3 &\leftarrow x_r \times y_i; \\
t_4 &\leftarrow x_i \times y_r; \\
z_r &\leftarrow t_1 - t_2; \\
z_i &\leftarrow t_3 + t_4 \quad .
\end{aligned}
$$

Furthermore, we would like to have a precision that exceeds what our hardware can natively support. For example, let us assume that we have a machine with native support for an integer type $-R < x < R$. In this case, we split each variable ζ into a low part plus a high part: $\zeta = \zeta^{(0)} + R\zeta^{(1)}$, $-R < \zeta^{(0)}, \zeta^{(1)} < R$. Now let us assume that our machine has a multiplication instruction $(c^{(0)}, c^{(1)}) \leftarrow$

$a \times b$ such that $-R < a, b, c^{(0)}, c^{(1)} < R$ and $ab = c^{(0)} + Rc^{(1)}$. For simplicity, let us further assume that our machine has n-ary addition instructions for $n = 2, 3, 4$: $(c^{(0)}, c^{(1)}) \leftarrow a_1 + \cdots + a_n$ such that $-R < a_1, \ldots, a_n, c^{(0)}, c^{(1)} < R$ and $a_1 + \cdots + a_n = c^{(0)} + Rc^{(1)}$. We can then have a suboptimal yet straightforward implementation of, say, $t_1 = t_1^{(0)} + Rt_1^{(1)} + R^2 t_1^{(2)} + R^3 t_1^{(3)} = (x_r^{(0)} + Rx_r^{(1)}) \times (y_r^{(0)} + Ry_r^{(1)}) = x_r \times y_r$ as follows.

$$
\begin{array}{ll}
(t_1^{(0)}, s_1^{(1)}) \leftarrow x_r^{(0)} \times y_r^{(0)}; & // t_1^{(0)} + Rs_1^{(1)} \\
(s_2^{(0)}, s_2^{(1)}) \leftarrow x_r^{(0)} \times y_r^{(1)}; & // Rs_2^{(0)} + R^2 s_2^{(1)} \\
(s_3^{(0)}, s_3^{(1)}) \leftarrow x_r^{(1)} \times y_r^{(0)}; & // Rs_3^{(0)} + R^2 s_3^{(1)} \\
(s_4^{(0)}, s_4^{(1)}) \leftarrow x_r^{(1)} \times y_r^{(1)}; & // R^2 s_4^{(0)} + R^3 s_4^{(1)} \\
(t_1^{(1)}, s_5^{(1)}) \leftarrow s_1^{(1)} + s_2^{(0)} + s_3^{(0)}; & // Rt_1^{(1)} + R^2 s_5^{(1)} \\
(t_1^{(2)}, s_6^{(1)}) \leftarrow s_2^{(1)} + s_3^{(1)} + s_4^{(0)} + s_5^{(1)}; & // R^2 t_1^{(2)} + R^3 s_6^{(1)} \\
(t_1^{(3)}, \_) \leftarrow s_4^{(1)} + s_6^{(1)} \quad . & // R^3 t_1^{(3)}
\end{array}
$$

It might be surprising that, with the advance of compiler technology today, such programs are still mostly coded and optimized manually, sometimes in assembly language, for maximal efficiency. Naturally, we would like to automate this process as much as possible. Furthermore, with such security-critical applications, we would like to have machine-verified proofs that the transformation and optimizations are correct.

In attempting toward this goal, we have come up with this pearl. It is not yet practical but, we think, is neat and worth documenting. A key observation is that there is an isomorphism between multivariate polynomial ring $R[X_1, X_2 \ldots, X_m]$ and univariate polynomial ring $S[X_m]$ over the base ring $S = R[X_1, X_2, \ldots, X_{m-1}]$, cf. Corollary 5.7, Chapter 3 of Hungerford [6]. This allows us to define a datatype Poly for univariate polynomials, and reuse it inductively to define multivariate polynomials — an n-variate polynomial can be represented by $\mathsf{Poly}^{\mathsf{n}}$ (Poly applied n times). Most operations on the polynomials can be defined either in terms of the *fold* induced by Poly, or by induction on n, hence the title.

We explore the use of $\mathsf{Poly}^{\mathsf{n}}$ and its various implications in depth in Section 2. Then in Section 3, we present implementations of common polynomial operations such as evaluation, substitution, etc. We pay special attention to an operation expand and prove its correctness, since it is essential in transforming polynomial into scalar expressions. In Section 4, we show how to compile a polynomial function into an assembly program that computes it. We present a simple compilation, also defined in terms of fold, and prove its correctness, while leaving further optimization to future work. The formulation in this pearl have been implemented in both Haskell and Agda [9], the latter also used to verify our proofs. The code is available on line at `https://github.com/petercommand/ExtFieldComp`.

2 Univariate and Multivariate Polynomials

In this section, we present our representation for univariate and multivariate polynomials, as well as their semantics. The following Agda datatype denotes a univariate polynomial whose coefficients are of type A:[4]

```
data Poly (A : Set) : Set where
  Ind  : Poly A
  Lit  : A → Poly A
  (:+) : Poly A → Poly A → Poly A
  (:×) : Poly A → Poly A → Poly A  ,
```

where Ind denotes the indeterminate, Lit denotes a constant (of type A), while (:+) and (:×) respectively denote addition and multiplication. A polynomial $2x^2 + 3x + 1$ can be represented by the following expression of type Poly $\mathbb{Z}$:

```
(Lit 2 :× Ind :× Ind) :+ (Lit 3 :× Ind) :+ Lit 1  .
```

Notice that the type parameter A is abstracted over the type of coefficients. This allows us to represent polynomials whose coefficients have non-simple types — in particular, polynomials whose coefficients are themselves polynomials. Do not confuse this with the more conventional representation of arithmetic expressions:

```
data Expr A = Var A | Lit Int | Expr A :+ Expr A | Expr A :× Expr A  ,
```

where the literals are usually assigned a fixed type (in this example, Int), and the type parameter is abstracted over variables Var.

2.1 Univariate Polynomial and its Semantics

In the categorical style outlined by Bird and de Moor [1], every *regular* datatype gives rise to a *fold*, also called a *catamorphism*. The type Poly induces a fold that, conventionally, takes four arguments, each replacing one of its four constructors. To facilitate our discussion later, we group the last two arguments together. The fold for Poly is thus given by:

```
foldP : {A B : Set} → B → (A → B) →
        ((B → B → B) × (B → B → B)) → Poly A → B
foldP x f ((⊕) , (⊗)) Ind          = x
foldP x f ((⊕) , (⊗)) (Lit y)      = f y
foldP x f ((⊕) , (⊗)) (e₁ :+ e₂)  = foldP x f ((⊕) , (⊗)) e₁ ⊕
                                      foldP x f ((⊕) , (⊗)) e₂
foldP x f ((⊕) , (⊗)) (e₁ :× e₂)  = foldP x f ((⊕) , (⊗)) e₁ ⊗
                                      foldP x f ((⊕) , (⊗)) e₂  ,
```

where arguments x, f, (⊕), and (⊗) respectively replace constructors Ind, Lit, (:+), and (:×).

[4] We use Haskell convention that infix data constructors start with a colon and, for concise typesetting, write (:+) instead of the Agda notation _:+_. In the rest of the paper we also occasionally use Haskell syntax for brevity.

Evaluation. To evaluate a polynomial of type $\mathsf{Poly\ A}$, we have to know how to perform arithmetic operations for type A. Define

$$\begin{aligned}&\mathsf{Ring} : \mathsf{Set} \to \mathsf{Set}\\ &\mathsf{Ring\ A} = ((\mathsf{A} \to \mathsf{A} \to \mathsf{A}) \times (\mathsf{A} \to \mathsf{A} \to \mathsf{A})) \times \mathsf{A} \times \mathsf{A} \times (\mathsf{A} \to \mathsf{A}) \quad ,\end{aligned}$$

the intention is that the tuple $\mathsf{Ring\ A}$ defines addition, multiplication, zero, one, and negation for A (addition and multiplication are grouped together, for our convenience later). In our Haskell implementation, Ring is a type class for types whose addition and multiplication are defined. It can usually be inferred what instance of Ring to use. When proving properties about foldP, however, it is clearer to make the construction of Ring instances explicit.

With the presence of Ind, the semantics of $\mathsf{Poly\ A}$ should be $\mathsf{A} \to \mathsf{A}$ — a function that takes the value of the indeterminate and returns a value. We define the following operation that lifts pointwise the addition and multiplication of some type B to $\mathsf{A} \to \mathsf{B}$:

$$\begin{aligned}&\mathsf{ring}_{\to} : \forall\ \{\mathsf{A\ B}\} \to \mathsf{Ring\ B} \to \mathsf{Ring\ (A \to B)}\\ &\mathsf{ring}_{\to}\ (((+)\ ,(\times))\ ,\mathbf{0},\mathbf{1},\mathsf{neg}) =\\ &\quad ((\lambda\ \mathsf{f\ g\ x} \to \mathsf{f\ x} + \mathsf{g\ x}, \lambda\ \mathsf{f\ g\ x} \to \mathsf{f\ x} \times \mathsf{g\ x})\ , \mathsf{const}\ \mathbf{0}, \mathsf{const}\ \mathbf{1}, (\mathsf{neg}\cdot)) \quad ,\end{aligned}$$

where $\mathsf{const\ x\ y} = \mathsf{x}$. The semantics of a univariate polynomial is thus given by:

$$\begin{aligned}&\mathsf{sem}_1 : \forall\ \{\mathsf{A}\} \to \mathsf{Ring\ A} \to \mathsf{Poly\ A} \to \mathsf{A} \to \mathsf{A}\\ &\mathsf{sem}_1\ \mathsf{rng} = \mathsf{foldP\ id\ const\ (fst\ (ring_{\to}\ rng))} \quad ,\end{aligned}$$

where $\mathsf{id\ x} = \mathsf{x}$ and fst retrieves the left component of a pair.

2.2 Bivariate Polynomials

To represent polynomials with two indeterminates, one might extend Poly with a constructor Ind' in addition to Ind. It turns out to be unnecessary — it is known that the bivariate polynomial ring $R[X,Y]$ is isomorphic to $R[X][Y]$ (modulo the operation $\mathsf{litDist}$, to be defined later). That is, a polynomial over base ring A with two indeterminates can be represented by $\mathsf{Poly\ (Poly\ A)}$.

To understand the isomorphism, consider the following expression:

$$\begin{aligned}&\mathsf{e} : \mathsf{Poly}\ (\mathsf{Poly}\ \mathbb{Z})\\ &\mathsf{e} = (\mathsf{Lit\ (Lit\ 3)} :\times \mathsf{Ind} :\times \mathsf{Lit\ (Ind} :+ \mathsf{Lit\ 4))} :+ \mathsf{Lit\ Ind} :+ \mathsf{Ind} \quad .\end{aligned}$$

Note that to represent a literal 3, we have to write $\mathsf{Lit\ (Lit\ 3)}$, since the first Lit takes a $\mathsf{Poly}\ \mathbb{Z}$ as its argument. To evaluate e using sem_1, we have to define $\mathsf{Ring}\ (\mathsf{Poly}\ \mathbb{Z})$. A natural choice is to connect two expressions using corresponding constructors:

$$\begin{aligned}&\mathsf{ringP} : \forall\ \{\mathsf{A}\} \to \mathsf{Ring\ A} \to \mathsf{Ring\ (Poly\ A)}\\ &\mathsf{ringP}\ (\_\ ,\mathbf{0}\ ,\mathbf{1}\ ,\mathsf{neg}) = (((:+)\ ,(:\times))\ ,\mathsf{Lit}\ \mathbf{0}\ ,\mathsf{Lit}\ \mathbf{1}\ ,(\mathsf{Lit}\ (\mathsf{neg}\ \mathbf{1}) :\times)) \quad .\end{aligned}$$

With ringP defined, $\mathsf{sem}_1\ (\mathsf{ringP}\ \mathsf{r})\ \mathsf{e}$ has type $\mathsf{Poly\ A} \to \mathsf{Poly\ A}$. Evaluating, for example $\mathsf{sem}_1\ (\mathsf{ringP}\ \mathsf{r})\ \mathsf{e}\ (\mathsf{Ind} :+ \mathsf{Lit}\ 1)$, yields

$$
\begin{array}{l}
\mathsf{e}' : \mathsf{Poly}\ \mathbb{Z} \\
\mathsf{e}' \ = \ (\mathsf{Lit}\ 3 :\times (\mathsf{Ind} :+ \mathsf{Lit}\ 1) :\times (\mathsf{Ind} :+ \mathsf{Lit}\ 4)) :+ \mathsf{Ind} :+ (\mathsf{Ind} :+ \mathsf{Lit}\ 1) \quad .
\end{array}
$$

Note that $\mathsf{Lit\ Ind}$ in e is replaced by the argument $\mathsf{Ind} :+ \mathsf{Lit}\ 1$. Furthermore, one layer of Lit is removed, thus both $\mathsf{Lit}\ 3$ and $\mathsf{Ind} :+ \mathsf{Lit}\ 4$ are exposed to the outermost level. The expression e' may then be evaluated by $\mathsf{sem}_1\ \mathsf{rng}\mathbb{Z}$, where $\mathsf{rng}\mathbb{Z} : \mathsf{Ring}\ \mathbb{Z}$. The result is a natural number. In general, the function sem_2 that evaluates $\mathsf{Poly\ (Poly\ A)}$ can be defined by:

$$
\begin{array}{l}
\mathsf{sem}_2 : \forall\ \{\mathsf{A}\} \to \mathsf{Ring\ A} \to \mathsf{Poly\ (Poly\ A)} \to \mathsf{Poly\ A} \to \mathsf{A} \to \mathsf{A} \\
\mathsf{sem}_2\ \mathsf{r}\ \mathsf{e}_2\ \mathsf{e}_1\ \mathsf{x} \ = \ \mathsf{sem}_1\ \mathsf{r}\ (\mathsf{sem}_1\ (\mathsf{ringP}\ \mathsf{r})\ \mathsf{e}_2\ \mathsf{e}_1)\ \mathsf{x} \quad .
\end{array}
$$

This is how $\mathsf{Poly}\ (\mathsf{Poly}\ \mathbb{Z})$ simulates bivariate polynomials: the two indeterminates are respectively represented by Ind and $\mathsf{Lit\ Ind}$. During evaluation, Ind can be instantiated to an expression arg of type $\mathsf{Poly}\ \mathbb{Z}$, while $\mathsf{Lit\ Ind}$ can be instantiated to a $\mathbb{Z}$. If arg contains Ind, it refers to the next indeterminate.

What about expressions like $\mathsf{Lit}\ (\mathsf{Ind} :+ \mathsf{Lit}\ 4)$? One can see that its semantics is the same as $\mathsf{Lit\ Ind} :+ \mathsf{Lit}\ (\mathsf{Lit}\ 4)$, the expression we get by pushing Lit to the leaves. In general, define the following function:

$$
\begin{array}{l}
\mathsf{litDist} : \forall\ \{\mathsf{A}\} \to \mathsf{Poly\ (Poly\ A)} \to \mathsf{Poly\ (Poly\ A)} \\
\mathsf{litDist} \ = \ \mathsf{foldP\ Ind}\ (\mathsf{foldP}\ (\mathsf{Lit\ Ind})\ (\mathsf{Lit} \cdot \mathsf{Lit})\ ((:+)\ ,\ (:\times)))\ ((:+)\ ,\ (:\times)) \quad .
\end{array}
$$

The function traverses through the given expression and, upon encountering a subtree $\mathsf{Lit\ e}$, lifts e to $\mathsf{Poly\ (Poly\ A)}$ while distributing Lit inwards e. We can prove the following theorem:

Theorem 1. *For all* $\mathsf{e} : \mathsf{Poly\ (Poly\ A)}$ *and* $\mathsf{r} : \mathsf{Ring\ A}$*, we have* $\mathsf{sem}_2\ \mathsf{r}\ (\mathsf{litDist\ e}) = \mathsf{sem}_2\ \mathsf{r}\ \mathsf{e}$*.*

2.3 Multivariate Polynomials

In general, as we have mentioned in Section 1, the multivariate polynomial $R[X_1, X_2 \ldots, X_m]$ is isomorphic to univariate polynomial ring $S[X_m]$ over the base ring $S = R[X_1, X_2, \ldots, X_{m-1}]$ (modulo the distribution law of Lit). That is, a polynomial over A with n indeterminates can be represented by $\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$, defined by

$$
\begin{array}{l}
\mathsf{Poly}^{\mathsf{zero}}\ \mathsf{A} \ = \ \mathsf{A} \\
\mathsf{Poly}^{\mathsf{suc\ n}}\ \mathsf{A} \ = \ \mathsf{Poly}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}) \quad .
\end{array}
$$

To define the semantics of $\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$, recall that, among its n indeterminates, the outermost indeterminate shall be instantiated to an expression of type $\mathsf{Poly}^{\mathsf{n}-1}\ \mathsf{A}$, the next indeterminate to $\mathsf{Poly}^{\mathsf{n}-2}\ \mathsf{A}$..., and the inner most indeterminate to A, before yielding a value of type A. Define

$$\begin{array}{lcl}
\mathsf{DChain} : \mathsf{Set} \to \mathbb{N} \to \mathsf{Set} & & \\
\mathsf{DChain}\ \mathsf{A}\ \mathsf{zero} & = & \top \\
\mathsf{DChain}\ \mathsf{A}\ (\mathsf{suc}\ \mathsf{n}) & = & \mathsf{Poly}^{\mathsf{n}}\ \mathsf{A} \times \mathsf{DChain}\ \mathsf{A}\ \mathsf{n} \quad ,
\end{array}$$

that is, $\mathsf{DChain}\ \mathsf{A}\ \mathsf{n}$ (the name standing for a "descending chain") is a list of n elements, with the first having type $\mathsf{Poly}^{\mathsf{n}-1}\ \mathsf{A}$, the second $\mathsf{Poly}^{\mathsf{n}-2}\ \mathsf{A}$, and so on. The type $\top$ denotes the "unit" type, inhabited by exactly one term tt.

Given an implementation of $\mathsf{Ring}\ \mathsf{A}$, the semantics of $\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$ is a function $\mathsf{DChain}\ \mathsf{A}\ \mathsf{n} \to \mathsf{A}$, defined inductively as below:

$$\begin{array}{lcl}
\mathsf{sem} : \forall\ \{\mathsf{A}\} \to \mathsf{Ring}\ \mathsf{A} \to (\mathsf{n} : \mathbb{N}) \to \mathsf{Poly}^{\mathsf{n}}\ \mathsf{A} \to \mathsf{DChain}\ \mathsf{A}\ \mathsf{n} \to \mathsf{A} & & \\
\mathsf{sem}\ \mathsf{r}\ \mathsf{zero}\quad \mathsf{x}\ \mathsf{tt} & = & \mathsf{x} \\
\mathsf{sem}\ \mathsf{r}\ (\mathsf{suc}\ \mathsf{n})\ \mathsf{e}\ (\mathsf{t}\ ,\ \mathsf{es}) & = & \mathsf{sem}\ \mathsf{r}\ \mathsf{n}\ (\mathsf{sem}_1\ (\mathsf{ringP}^{\star}\ \mathsf{r}\ \mathsf{n})\ \mathsf{e}\ \mathsf{t})\ \mathsf{es} \quad ,
\end{array}$$

where $\mathsf{ringP}^{\star}$ delivers the $\mathsf{Ring}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})$ instance for all n:

$$\begin{array}{lcl}
\mathsf{ringP}^{\star} : \forall\ \{\mathsf{A}\} \to \mathsf{Ring}\ \mathsf{A} \to \forall\ \mathsf{n} \to \mathsf{Ring}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}) & & \\
\mathsf{ringP}^{\star}\ \mathsf{r}\ \mathsf{zero} & = & \mathsf{r} \\
\mathsf{ringP}^{\star}\ \mathsf{r}\ (\mathsf{suc}\ \mathsf{n}) & = & \mathsf{ringP}\ (\mathsf{ringP}^{\star}\ \mathsf{r}\ \mathsf{n}) \quad .
\end{array}$$

For $\mathsf{n} := 2$ and 3, for example, $\mathsf{sem}\ \mathsf{r}\ \mathsf{n}$ expands to:

$$\begin{array}{lcl}
\mathsf{sem}\ \mathsf{r}\ 2\ \mathsf{e}\ (\mathsf{t}_1,\ \mathsf{t}_0,\ \mathsf{tt}) & = & \mathsf{sem}_1\ \mathsf{r}\ (\mathsf{sem}_1\ (\mathsf{ringP}\ \mathsf{r})\ \mathsf{e}\ \mathsf{t}_1)\ \mathsf{t}_0 \\
 & = & (\mathsf{sem}_1\ \mathsf{r} \cdot \mathsf{sem}_1\ (\mathsf{ringP}\ \mathsf{r})\ \mathsf{e})\ \mathsf{t}_1\ \mathsf{t}_0 \quad , \\
\mathsf{sem}\ \mathsf{r}\ 3\ \mathsf{e}\ (\mathsf{t}_2,\ \mathsf{t}_1,\ \mathsf{t}_0,\ \mathsf{tt}) & = & \mathsf{sem}_1\ \mathsf{r}\ (\mathsf{sem}_1\ (\mathsf{ringP}\ \mathsf{r})\ (\mathsf{sem}_1\ (\mathsf{ringP}^2\ \mathsf{r})\ \mathsf{e}\ \mathsf{t}_2)\ \mathsf{t}_1)\ \mathsf{t}_0 \\
 & = & (\mathsf{sem}_1\ \mathsf{r} \cdot \mathsf{sem}_1\ (\mathsf{ringP}\ \mathsf{r}) \cdot \mathsf{sem}_1\ (\mathsf{ringP}^2\ \mathsf{r})\ \mathsf{e})\ \mathsf{t}_2\ \mathsf{t}_1\ \mathsf{t}_0 \quad .
\end{array}$$

Essentially, $\mathsf{sem}\ \mathsf{r}\ \mathsf{n}$ is n-fold composition of $\mathsf{sem}_1\ (\mathsf{ringP}^{i}\ \mathsf{r})$, each interpreting one level of the given expression.

3 Operations on Polynomials

Having defined a representation for multivariate polynomials, we ought to demonstrate that this representation is feasible — that we can define most of the operations we want. In fact, it turns that most of them can be defined either in terms of foldP or by induction over the number of iterations Poly is applied.

3.1 Rotation

The first operation swaps the two outermost indeterminates of a $\mathsf{Poly}^2\ \mathsf{A}$, using foldP. This function witnesses the isomorphism between $R[X_1, \ldots, X_{m-1}][X_m]$ and $R[X_m, X_1, \ldots, X_{m-2}][X_{m-1}]$. It is instructive comparing it with $\mathsf{litDist}$.

$$\begin{array}{lcl}
\mathsf{rotaPoly}_2 : \forall\ \{\mathsf{A}\} \to \mathsf{Poly}^2\ \mathsf{A} \to \mathsf{Poly}^2\ \mathsf{A} & & \\
\mathsf{rotaPoly}_2 & = & \mathsf{foldP}\ (\mathsf{Lit}\ \mathsf{Ind})\ (\mathsf{foldP}\ \mathsf{Ind}\ (\mathsf{Lit} \cdot \mathsf{Lit})\ ((:+)\ ,\ (:\times)))\ ((:+)\ ,\ (:\times)) \quad .
\end{array}$$

In $\mathsf{rotaPoly}_2$, the outermost Ind is replaced by $\mathsf{Lit}\ \mathsf{Ind}$. When encountering $\mathsf{Lit}\ \mathsf{e}$, the inner e is lifted to $\mathsf{Poly}^2\ \mathsf{A}$. The Ind inside e remains Ind, which becomes the outermost indeterminate after lifting. Note that both $\mathsf{litDist}$ and $\mathsf{rotaPoly}_2$ apply to $\mathsf{Poly}^\mathsf{n}\ \mathsf{A}$ for all $\mathsf{n} \geqslant 2$, since A can be instantiated to a polynomial as well.

Consider $\mathsf{Poly}^3\ \mathsf{A}$, a polynomial with (at least) three indeterminates. To "rotate" the three indeterminates, that is, turn $\mathsf{Lit}^2\ \mathsf{Ind}$ to $\mathsf{Lit}\ \mathsf{Ind}$, $\mathsf{Lit}\ \mathsf{Ind}$ to Ind, and Ind to $\mathsf{Lit}^2\ \mathsf{Ind}$, we can define:

$$\mathsf{rotaPoly}_3 \ = \ \mathsf{fmap}\ \mathsf{rotaPoly}_2 \cdot \mathsf{rotaPoly}_2 \quad ,$$

where fmap is the usual "functorial map" function for Poly:

$$\mathsf{fmap} : \forall\ \{\mathsf{A}\ \mathsf{B}\} \to (\mathsf{A} \to \mathsf{B}) \to \mathsf{Poly}\ \mathsf{A} \to \mathsf{Poly}\ \mathsf{B} \quad .$$

The first $\mathsf{rotaPoly}_2$ swaps the two outer indeterminates, while $\mathsf{fmap}\ \mathsf{rotaPoly}_2$ swaps the inner two. To rotate the outermost four indeterminates of a $\mathsf{Poly}^4\ \mathsf{A}$, we may define:

$$\mathsf{rotaPoly}_4 \ = \ \mathsf{fmap}\ (\mathsf{fmap}\ \mathsf{rotaPoly}_2) \cdot \mathsf{rotaPoly}_3 \quad .$$

In general, the following function rotates the first n indeterminates of the given polynomial:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathsf{rotaPoly} : \forall\ \{\mathsf{A}\}\ (\mathsf{n} : \mathbb{N}) \to \mathsf{Poly}^\mathsf{n}\ \mathsf{A} \to \mathsf{Poly}^\mathsf{n}\ \mathsf{A}} \\
\mathsf{rotaPoly}\ \mathsf{zero} & = & \mathsf{id} \\
\mathsf{rotaPoly}\ (\mathsf{suc}\ \mathsf{zero}) & = & \mathsf{id} \\
\mathsf{rotaPoly}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})) & = & \mathsf{rotaPoly}_2 \\
\mathsf{rotaPoly}\ (\mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{n}))) & = & \mathsf{fmap}^{\mathsf{suc}\ \mathsf{n}}\ \mathsf{rotaPoly}_2 \cdot \mathsf{rotaPoly}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{n})) \quad .
\end{array}$$

Note that in the actual code we need to convince Agda that $\mathsf{Poly}^\mathsf{n}\ (\mathsf{Poly}\ \mathsf{A})$ is the same type as $\mathsf{Poly}\ (\mathsf{Poly}^\mathsf{n}\ \mathsf{A})$ and use subst to coerce between the two types. We omit those details for clarity.

Given m and n, $\mathsf{rotaOuter}\ \mathsf{n}\ \mathsf{m}$ compose $\mathsf{rotaPoly}\ \mathsf{n}$ with itself m times. Therefore, the outermost n indeterminates are rotated m times. It will be handy in Section 3.2.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathsf{rotaOuter} : \forall\ \{\mathsf{A}\}\ (\mathsf{n}\ \mathsf{m} : \mathbb{N}) \to \mathsf{Poly}^\mathsf{n}\ \mathsf{A} \to \mathsf{Poly}^\mathsf{n}\ \mathsf{A}} \\
\mathsf{rotaOuter}\ \mathsf{n}\ \mathsf{zero} & = & \mathsf{id} \\
\mathsf{rotaOuter}\ \mathsf{n}\ (\mathsf{suc}\ \mathsf{m}) & = & \mathsf{rotaOuter}\ \mathsf{n}\ \mathsf{m} \cdot \mathsf{rotaPoly}\ \mathsf{n}\ \mathsf{e} \quad .
\end{array}$$

3.2 Substitution

Substitution is another operation that one would expect. Given an expression e, how do we substitute, for each occurrence of Ind, another expression e', using operations we have defined? Noticing that the type of sem_1 can be instantiated to $\mathsf{Poly}^2\ \mathsf{A} \to \mathsf{Poly}\ \mathsf{A} \to \mathsf{Poly}\ \mathsf{A}$, we may lift e to $\mathsf{Poly}^2\ \mathsf{A}$ by wrapping it with Lit, do a $\mathsf{rotaPoly}_2$ to swap the Ind in e to the outermost position, and use sem_1 to perform the substitution:

$$\begin{aligned}&\mathsf{substitute}_1 : \forall\, \{\mathsf{A}\} \to \mathsf{Ring\ A} \to \mathsf{Poly\ A} \to \mathsf{Poly\ A} \to \mathsf{Poly\ A}\\ &\mathsf{substitute}_1\ \mathsf{r\ e\ e'}\ =\ \mathsf{sem}_1\ (\mathsf{ringP\ r})\ (\mathsf{rotaPoly}_2\ (\mathsf{Lit\ e}))\ \mathsf{e'}\quad .\end{aligned}$$

What about $\mathsf{e} : \mathsf{Poly}^2\ \mathsf{A}$? We may lift it to $\mathsf{Poly}^4\ \mathsf{A}$, perform two $\mathsf{rotaPoly}_4$ to expose its two indeterminates, before using sem_2:

$$\begin{aligned}&\mathsf{substitute}_2 :: \forall\, \{\mathsf{A}\} \to \mathsf{Ring\ A} \to \mathsf{Poly}^2\ \mathsf{A} \to \mathsf{Poly}^2\ \mathsf{A} \to \mathsf{Poly}^2\ \mathsf{A} \to \mathsf{Poly}^2\ \mathsf{A}\\ &\mathsf{substitute}_2\ \mathsf{r\ e\ e'\ e''}\ =\\ &\quad \mathsf{sem}_2\ (\mathsf{ringP}\ (\mathsf{ringP\ r}))\ (\mathsf{rotaPoly}_4\ (\mathsf{rotaPoly}_4\ \mathsf{Lit}\ (\mathsf{Lit\ e})))\ (\mathsf{Lit\ e'})\ \mathsf{e''}\quad .\end{aligned}$$

Consider the general case with substituting the n indeterminates in $\mathsf{e} : \mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$ for n expressions, each of type $\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$. Let $\mathsf{Vec\ B\ n}$ be the type of vectors (lists of fixed lengths) of length n. A general $\mathsf{substitute}$ can be defined by:

$$\begin{aligned}&\mathsf{substitute} : \forall\, \{\mathsf{A}\}\ \mathsf{n} \to \mathsf{Ring\ A} \to \mathsf{Poly}^{\mathsf{n}}\ \mathsf{A} \to \mathsf{Vec}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})\ \mathsf{n} \to \mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}\\ &\mathsf{substitute}\ \{\mathsf{A}\}\ \mathsf{n\ r\ e\ es}\ =\\ &\quad \mathsf{sem}\ (\mathsf{ringP}^{\star}\ \mathsf{r\ n})\ \mathsf{n}\ (\mathsf{rotaOuter}\ (\mathsf{n}+\mathsf{n})\ \mathsf{n}\ (\mathsf{liftPoly\ n}\ (\mathsf{n}+\mathsf{n})\ \mathsf{e}))\\ &\qquad\qquad\qquad (\mathsf{toDChain\ es})\quad ,\end{aligned}$$

where $\mathsf{liftPoly\ n\ m}$ (with $\mathsf{n} \leqslant \mathsf{m}$) lifts a $\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$ to $\mathsf{Poly}^{\mathsf{m}}\ \mathsf{A}$ by applying Lit; $\mathsf{rotaOuter}\ (\mathsf{n}+\mathsf{n})\ \mathsf{n}$, as defined in Section 3.1, composes $\mathsf{rotaPoly}\ (\mathsf{n}+\mathsf{n})$ with itself n times, thereby moving the n original indeterminates of e to outermost positions; the function $\mathsf{toDChain} : \forall\, \{\mathsf{A}\}\ \mathsf{n} \to \mathsf{Vec\ A\ n} \to \mathsf{DChain\ A\ n}$ converts a vector to a descending chain, informally,

$$\mathsf{toDChain}\ [\ \mathsf{t}_2,\ \mathsf{t}_1,\ \mathsf{t}_0\ ]\ =\ (\mathsf{Lit}\ (\mathsf{Lit\ t}_2)\ ,\ \mathsf{Lit\ t}_1,\ \mathsf{t}_0,\ \mathsf{tt})\quad ;$$

finally, sem performs the substitution. Again, the actual code needs additional proof terms (to convince Agda that $\mathsf{n} \leqslant \mathsf{n}+\mathsf{n}$) and type coercion (between $\mathsf{Poly}^{\mathsf{n}}\ (\mathsf{Poly}^{\mathsf{m}}\ \mathsf{A})$ and $\mathsf{Poly}^{\mathsf{m}+\mathsf{n}}\ \mathsf{A}$), which are omitted here.

3.3 Expansion

Expansion is an operation we put specific emphasis on, since it is useful when implementing cryptosystems on microprocessors with no native hardware support for arithmetic operations with polynomials or integers of cryptographic sizes. Let us use a simple yet specific example for further exposition: the polynomial expression over complex numbers $(3+2i)x^2 + (2+i)x + 1$ can be represented by $\mathsf{Poly}\ (\mathbb{R} \times \mathbb{R})$, whose semantics is a function $(\mathbb{R} \times \mathbb{R}) \to (\mathbb{R} \times \mathbb{R})$. Let x be $x_1 + x_2 i$, the polynomial can be expanded as below:

$$\begin{aligned}&\ (3+2i)(x_1+x_2 i)^2 + (2+i)(x_1+x_2 i) + 1\\ =&\ (3x_1^2 - 4x_1x_2 - 3x_2^2) + (2x_1^2 + 6x_1x_2 - 2x_2^2)i + (2x_1 - x_2) + (x_1 + 2x_2)i + 1\\ =&\ (3x_1^2 + 2x_1 - 4x_1x_2 - x_2 - 3x_2^2 + 1) + (2x_1^2 + x_1 + 6x_1x_2 + 2x_2 - 2x_2^2)i\quad .\end{aligned}$$

That is, a univariate polynomial over pairs, $\mathsf{Poly}\ (\mathbb{R} \times \mathbb{R})$, can be expanded to $(\mathsf{Poly}^2\ \mathbb{R} \times \mathsf{Poly}^2\ \mathbb{R})$, a pair of bivariate expressions. The expansion depends on the definitions of addition and multiplication of complex numbers.

It might turn out that $\mathbb{R}$ is represented by a fixed number of machine words: $\mathbb{R} = \mathsf{Word}^{\mathsf{n}}$. As mentioned before, in cryptosystems n could be hundreds. To compute the value of the polynomial, $\mathsf{Poly}\ \mathsf{Word}^{\mathsf{n}}$ can be further expanded to $(\mathsf{Poly}^{\mathsf{n}}\ \mathsf{Word})^{\mathsf{n}}$, this time using arithmetic operations defined for Word. Now that each polynomial is defined over Word, whose arithmetic operations are natively supported, we may compile the expressions, in ways discussed in Section 4, into a sequence of operations in assembly language. We also note that the roles played by the indeterminates x and i are of fundamental difference: x might just represent the input of the computation modelled by the polynomial expression, which will be substituted by some values at runtime, whereas i intends to model some internal (algebraic) structures and is never substituted throughout the whole computation.

Currently, such conversion and compilation are typically done by hand. We define expansion in this section and compilation in the next, as well as proving their correctness.

In general, a univariate polynomial over n-vectors, $\mathsf{Poly}\ (\mathsf{Vec}\ \mathsf{A}\ \mathsf{n})$, can be expanded to a n-vector of n-variate polynomial, $\mathsf{Vec}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})\ \mathsf{n}$. To formally define expansion we need some helper functions. Firstly, $\mathsf{genInd}\ \mathsf{n}$ generates a vector $\mathsf{Ind} :: \mathsf{Lit}\ \mathsf{Ind} :: \ldots\ \mathsf{Lit}^{\mathsf{n}-1}\ \mathsf{Ind} :: [\,]$. It corresponds to expanding x to (x_1, x_2) in the previous example.

$$
\begin{array}{l}
\mathsf{genInd} : \forall\ \{\mathsf{A}\}\ \mathsf{n} \to \mathsf{Vec}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})\ \mathsf{n} \\
\begin{array}{lcl}
\mathsf{genInd}\ \mathsf{zero} & = & [\,] \\
\mathsf{genInd}\ (\mathsf{suc}\ \mathsf{zero}) & = & \mathsf{Ind} :: [\,] \\
\mathsf{genInd}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{n})) & = & \mathsf{Ind} :: \mathsf{map}\ \mathsf{Lit}\ (\mathsf{genInd}\ (\mathsf{suc}\ \mathsf{n}))\quad .
\end{array}
\end{array}
$$

Secondly, $\mathsf{liftVal} : \forall\ \{\mathsf{A}\}\ \mathsf{n} \to \mathsf{A} \to \mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$ lifts A to $\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$ by n applications of Lit. The definition is routine.

Expansion can now be defined by:

$$
\begin{array}{l}
\mathsf{expand} : \forall\ \{\mathsf{A}\}\ \mathsf{n} \to \mathsf{Ring}\ (\mathsf{Vec}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})\ \mathsf{n}) \to \mathsf{Poly}\ (\mathsf{Vec}\ \mathsf{A}\ \mathsf{n}) \to \mathsf{Vec}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})\ \mathsf{n} \\
\mathsf{expand}\ \mathsf{n}\ \mathsf{rv}\ =\ \mathsf{foldP}\ (\mathsf{genInd}\ \mathsf{n})\ (\mathsf{map}\ (\mathsf{liftVal}\ \mathsf{n}))\ (\mathsf{fst}\ \mathsf{rv})
\end{array}
$$

For the Ind case, one indeterminant is expanded to n using genInd. For the $\mathsf{Lit}\ \mathsf{xs}$ case, $\mathsf{xs} : \mathsf{Vec}\ \mathsf{A}\ \mathsf{n}$ can be lifted to $\mathsf{Vec}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})\ \mathsf{n}$ by $\mathsf{map}\ (\mathsf{liftVal}\ \mathsf{n})$. For addition and multiplication, we let rv decide how to combine vectors of expressions.

The function expand alone does not say much — all the complex work is done in $\mathsf{rv} : \mathsf{Ring}\ (\mathsf{Vec}\ (\mathsf{Poly}^{\mathsf{n}}\ \mathsf{A})\ \mathsf{n})$. To generate rv, we define the type of operations that, given arithmetic operators for A, define ring instance for vectors of A:

$$
\begin{array}{l}
\mathsf{RingVec} : \mathbb{N} \to \mathsf{Set}_1 \\
\mathsf{RingVec}\ \mathsf{n}\ =\ \forall\ \{\mathsf{A}\} \to \mathsf{Ring}\ \mathsf{A} \to \mathsf{Ring}\ (\mathsf{Vec}\ \mathsf{A}\ \mathsf{n})\quad .
\end{array}
$$

For example, $\mathsf{rComplex}$ lifts arithmetic operations on A to that of complex numbers over A:

$$
\begin{array}{l}
\mathsf{rComplex} : \mathsf{RingVec}\ 2 \\
\mathsf{rComplex}\ ((+)\ ,\ (\times)\ ,\ \mathbf{0}\ ,\ \mathbf{1}\ ,\ \mathsf{neg})\ =\ ((+_c)\ ,\ (\times_c)\ ,\ [\mathbf{0}, \mathbf{0}], [\mathbf{1}\ ,\ \mathbf{0}], \mathsf{negC})
\end{array}
$$

$$\begin{aligned}
\textbf{where}\ & [\,\mathsf{x}_1,\ \mathsf{y}_1\,] +_c [\,\mathsf{x}_2,\ \mathsf{y}_2\,] \;=\; [\,\mathsf{x}_1 + \mathsf{x}_2,\ \mathsf{y}_1 + \mathsf{y}_2\,] \\
& [\,\mathsf{x}_1,\ \mathsf{y}_1\,] \times_c [\,\mathsf{x}_2,\ \mathsf{y}_2\,] \;=\; [\,\mathsf{x}_1 \times \mathsf{x}_2 - \mathsf{y}_1 \times \mathsf{y}_2\ , \mathsf{x}_1 \times \mathsf{y}_2 + \mathsf{x}_2 \times \mathsf{y}_1\,] \\
& \mathsf{x} - \mathsf{y} \;=\; \mathsf{x} + \mathsf{neg}\ \mathbf{1} \times \mathsf{y} \\
& \mathsf{negC}\ [\,\mathsf{x}, \mathsf{y}\,] \;=\; [\,\mathsf{neg}\ \mathbf{1} \times \mathsf{x}_1,\ \mathsf{neg}\ \mathbf{1} \times \mathsf{y}\,] \quad .
\end{aligned}$$

To expand a polynomial of complex numbers $\mathsf{Poly}\ (\mathsf{Vec}\ \mathsf{A}\ 2)$, expand demands an instance of $\mathsf{Ring}\ (\mathsf{Vec}\ (\mathsf{Poly}^2\ \mathsf{A})\ 2)$. One may thus call $\mathsf{expand}\ 2\ (\mathsf{rComplex}\ (\mathsf{ringP}^2\ \mathsf{r})$, where $\mathsf{r} : \mathsf{Ring}\ \mathsf{A}$. That is, we use $\mathsf{rComplex}$ to combine a pair of polynomials, designating $((:+)\ , (:\times))$ as addition and multiplication.

Correctness. Intuitively, expand is correct if the expanded polynomial evaluates to the same value as that of the original. To formally state the property, we have to properly supply all the needed ingredients. Consider $\mathsf{e} : \mathsf{Poly}\ (\mathsf{Vec}\ \mathsf{A}\ \mathsf{n})$ — a polynomial whose coefficients are vectors of length n. Let $\mathsf{r} : \mathsf{Ring}\ \mathsf{A}$ define arithmetic operators for A, and let $\mathsf{ringVec} : \mathsf{RingVec}\ \mathsf{n}$ define how arithmetic operators for elements are lifted to vectors. We say that expand is correct if, for all $\mathsf{xs} : \mathsf{Vec}\ \mathsf{A}\ \mathsf{n}$:

$$\begin{aligned}
\mathsf{sem}_1\ (\mathsf{ringVec}\ \mathsf{r})\ \mathsf{e}\ \mathsf{xs} \;=\; & \mathsf{map}\ (\lambda\ \mathsf{e} \to \mathsf{sem}\ \mathsf{r}\ \mathsf{n}\ \mathsf{e}\ (\mathsf{toDChain}\ \mathsf{xs})) \\
& \qquad (\mathsf{expand}\ \mathsf{n}\ (\mathsf{ringVec}\ (\mathsf{ringP}^\star\ \mathsf{r}\ \mathsf{n}))\ \mathsf{e}) \quad .
\end{aligned} \tag{1}$$

On the lefthand side, e is evaluated by sem_1, using operators supplied by $\mathsf{ringVec}\ \mathsf{r}$. The value of the single indeterminant is $\mathsf{xs} : \mathsf{Vec}\ \mathsf{A}\ \mathsf{n}$, and the result also has type $\mathsf{Vec}\ \mathsf{A}\ \mathsf{n}$. On the righthand side, e is expanded to $\mathsf{Vec}\ (\mathsf{Poly}^\mathsf{n}\ \mathsf{A})\ \mathsf{n}$, for which we need an instance of $\mathsf{Ring}\ (\mathsf{Vec}\ (\mathsf{Poly}^\mathsf{n}\ \mathsf{A})\ \mathsf{n})$, generated by $\mathsf{ringVec}\ (\mathsf{ringP}^\star\ \mathsf{r}\ \mathsf{n})$. Each polynomial in the vector is then evaluated individually by $\mathsf{sem}\ \mathsf{r}\ \mathsf{n}$. The function $\mathsf{toDChain}$ converts a vector to a descending chain. The n elements in xs thus substitutes the n indeterminants of the expanded polynomial.

Interestingly, it turns out that expand is correct if $\mathsf{ringVec}$ is polymorphic — that is, the way it computes vectors out of vectors depends only on the shape of its inputs, regardless of the type and values of their elements.

Theorem 2. *For all* n*,* $\mathsf{e} : \mathsf{Poly}\ (\mathsf{Vec}\ \mathsf{A}\ \mathsf{n})$*,* $\mathsf{xs} : \mathsf{Vec}\ \mathsf{A}\ \mathsf{n}$*,* $\mathsf{r} : \mathsf{Ring}\ \mathsf{A}$*, and* $\mathsf{ringVec} : \mathsf{RingVec}$*, property* (1) *holds if* $\mathsf{ringVec}$ *is polymorphic.*

Proof. Induction on e. For the base cases we need two lemmas:

- for all n, x, $\mathsf{es} : \mathsf{DChain}\ \mathsf{A}\ \mathsf{n}$, and r, we have $\mathsf{sem}\ \mathsf{r}\ \mathsf{n}\ (\mathsf{liftVal}\ \mathsf{n}\ \mathsf{x})\ \mathsf{es} \;=\; \mathsf{x}$;
- for all n, $\mathsf{xs} : \mathsf{Vec}\ \mathsf{A}\ \mathsf{n}$, and $\mathsf{r} : \mathsf{Ring}\ \mathsf{A}$, we have
  $\mathsf{map}\ (\lambda\ \mathsf{e} \to \mathsf{sem}\ \mathsf{r}\ \mathsf{n}\ \mathsf{e}\ (\mathsf{toDChain}\ \mathsf{xs}))\ (\mathsf{genInd}\ \mathsf{n}) \;=\; \mathsf{xs}$.

The inductive case where $\mathsf{e} := \mathsf{e}_1 :+ \mathsf{e}_2$ eventually comes down to proving that (abbreviating $\lambda\ \mathsf{e} \to \mathsf{sem}\ \mathsf{r}\ \mathsf{n}\ \mathsf{e}\ (\mathsf{toDChain}\ \mathsf{xs})$ to sem'):

$$\begin{aligned}
& \mathsf{map}\ \mathsf{sem}'\ (\mathsf{expand}\ \mathsf{ringVec}\ \mathsf{n}\ \mathsf{e}_1) +_{\mathsf{VA}} \mathsf{map}\ \mathsf{sem}'\ (\mathsf{expand}\ \mathsf{ringVec}\ \mathsf{n}\ \mathsf{e}_2) \;= \\
& \quad \mathsf{map}\ \mathsf{sem}'\ (\mathsf{expand}\ \mathsf{ringVec}\ \mathsf{n}\ \mathsf{e}_1 +_{\mathsf{VP}} \mathsf{expand}\ \mathsf{ringVec}\ \mathsf{n}\ \mathsf{e}_2)
\end{aligned}$$

where $(+_{\mathsf{VA}}) \;=\; \mathsf{fst}\ (\mathsf{fst}\ (\mathsf{ringVec}\ \mathsf{r}))$ defines addition on vectors of A's, and $(+_{\mathsf{VP}}) \;=\; \mathsf{fst}\ (\mathsf{fst}\ (\mathsf{ringVec}\ (\mathsf{ringP}^\star\ \mathsf{r}\ \mathsf{n})))$ on vectors of polynomials. But this is implied by the free theorem of $\mathsf{ringVec}$. Specifically, $\mathsf{fst} \cdot \mathsf{fst} \cdot \mathsf{ringVec}$ has type

$$\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Ring\ A} \to (\mathsf{Vec\ A\ n} \to \mathsf{Vec\ A\ n} \to \mathsf{Vec\ A\ n}) \quad .$$

The free theorem it induces is

$$\begin{array}{l}
\forall\ (\mathsf{X\ Y} : \mathsf{Set})\ \mathsf{n} \to \\
\quad \forall\ (\mathsf{f} : \mathsf{X} \to \mathsf{Y})\ (\mathsf{ring}_1 : \mathsf{Ring\ X})\ (\mathsf{ring}_2 : \mathsf{Ring\ Y}) \to \mathsf{P\ f\ ring}_1\ \mathsf{ring}_2 \to \\
\quad \forall\ (\mathsf{xs\ ys} : \mathsf{Vec\ X\ n}) \to \\
\quad\quad \textbf{let}\ (+_{\mathsf{V1}}) \ = \ \mathsf{fst\ (fst\ (ringVec\ ring}_1))\\
\quad\quad\quad\ \ (+_{\mathsf{V2}}) \ = \ \mathsf{fst\ (fst\ (ringVec\ ring}_2))\\
\quad\quad \textbf{in}\ \mathsf{map\ f\ (xs} +_{\mathsf{V1}} \mathsf{ys)} \ = \ \mathsf{map\ f\ xs} +_{\mathsf{V2}} \mathsf{map\ f\ ys} \quad ,
\end{array}$$

where P is given by:

$$\begin{array}{l}
\mathsf{P\ f}\ ((+_1)\ ,\ (\times_1)\ ,\ \mathbf{0}_1\ ,\ \mathbf{1}_1\ ,\ \mathsf{neg}_1)\ ((+_2)\ ,\ (\times_2)\ ,\ \mathbf{0}_2\ ,\ \mathbf{1}_2\ ,\ \mathsf{neg}_2) \ = \\
\quad \forall\ \mathsf{x}_1\ \mathsf{x}_2\ \mathsf{y}_1\ \mathsf{y}_2 \to \mathsf{f}\ (\mathsf{x}_1 +_1 \mathsf{x}_2) \ = \ \mathsf{f\ x}_1 +_2 \mathsf{f\ x}_2 \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad\ \ \mathsf{f}\ (\mathsf{x}_1 \times_1 \mathsf{x}_2) \ = \ \mathsf{f\ x}_1 \times_2 \mathsf{f\ x}_2 \ \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad\ \ \mathsf{f}\ \mathbf{0}_1 \ = \ \mathbf{0}_2 \ \wedge\ \mathsf{f}\ \mathbf{1}_1 \ = \ \mathbf{1}_2 \ \wedge\ \mathsf{f}\ (\mathsf{neg}_1\ \mathsf{x}_1) \ = \ \mathsf{neg}_2\ (\mathsf{f\ x}_2) \quad .
\end{array}$$

The conclusion of the free theorem is exactly what we need, while proving the premise is routine. The case for $\mathsf{e} := \mathsf{e}_1 :\times \mathsf{e}_2$ is similar. □

4 Compiling Polynomials

A potentially complex polynomial can be expanded, in several passes, to a vector of polynomial over Word, which can be compiled separately. As we have mentioned, such compilation is useful for software implementation of cryptosystems. Furthermore, even for hardware implementation, such compilation can be useful, as we can break down a complicated polynomial expression into a sequence of simpler arithmetic operations in a smaller algebraic structure, reducing the design complexity.

We consider a simple imaginary machine with a heap, denoted by Heap, that may abstractly be seen as mapping between memory addresses Addr and machine words Word. Albeit its simplicity, we believe that such a model captures the essential aspects of a wide variety of hardware and software implementations. The operator $(!!) : \mathsf{Heap} \to \mathsf{Addr} \to \mathsf{Word}$ fetches the value stored in the given address, while $\mathsf{ringWord} : \mathsf{Ring\ Word}$ defines how words are added and multiplied. The simple assembly language we consider consists of three instructions:

```
data TAC : Set where
  Const : Addr → Word → TAC
  Add : Addr → Addr → Addr → TAC
  Mul : Addr → Addr → Addr → TAC   ,

Ins : Set
Ins = List TAC   .
```

The command $\mathsf{Const\ i\ v}$ stores value v in address i, $\mathsf{Add\ i\ j\ k}$ fetches values stored in addresses i and j and stores their sum in address k, and similarly with Mul. Given a heap, executing an assembly program computes a new heap:

$$\mathsf{runIns} : \mathsf{Heap} \to \mathsf{Ins} \to \mathsf{Heap} \quad .$$

To compile a program we employ a monad SSA, which support an operation alloc : SSA Addr that returns the address of an unused cell in the heap. A naive approach is to implement SSA by a state monad that keeps a counter of the highest address that is allocated, while alloc returns the current value of the counter before incrementing it — register allocation can be performed in a separate pass. To run a SSA monad we use a function runSSA : $\forall \{\mathsf{A}\ \mathsf{St}\} \to \mathsf{St} \to \mathsf{SSA}\ \mathsf{St}\ \mathsf{A} \to (\mathsf{A} \times \mathsf{St})$ that takes a state St and yields a pair containing the result and the new state.

Compilation of a polynomial yields SSA (Addr × Ins), where the second component of the pair is an assembly program, and the first component is the address where the program, once run, stores the value of the polynomial. We define compilation of $\mathsf{Poly}^{\mathsf{n}}$ Word by induction on n. For the base case $\mathsf{Poly}^{0}\ \mathsf{Word} = \mathsf{Word}$, we simply allocate a new cell and store the given value there using Const:

```
compile₀ : Word → SSA (Addr × Ins)
compile₀ v = alloc ≫= λ addr →
               return (addr , Const addr v :: [ ])  .
```

To compile a polynomial of type $\mathsf{Poly}^{\mathsf{n}}$ Word, we assume that the value of the n indeterminants are already computed and stored in the heap, the locations of which are stored in a vector of n addresses.

```
compile : ∀ n → Vec Addr n → Polyⁿ Word → SSA (Addr × Ins)
compile zero     addr         = compile₀
compile (suc n) (x :: addr) =
  foldP (return (x, [ ])) (compile n addr) (biOp Add, biOp Mul)  .
```

In the clause for suc n, x is the address storing the value for the outermost indeterminant. To compile Ind, we simply return this address without generating any code. To compile Lit e where e : $\mathsf{Poly}^{\mathsf{n}}$ Word, we inductively call compile n addr. The generated code is combined by biOp op p_1 p_2, which runs p_1 and p_2 to obtain the compiled code, allocate a new address dest, before generating a new instruction op dest addr_1 addr_2:

```
biOp : (Addr → Addr → Addr → TAC)
       → SSA (Addr × Ins) → SSA (Addr × Ins) → SSA (Addr × Ins)
biOp op m₁ m₂ = m₁ ≫= λ (addr₁ , ins₁) →
                  m₂ ≫= λ (addr₂ , ins₂) → alloc ≫= λ dest →
                  return (dest , ins₁ ++ ins₂ ++ (op dest addr₁ addr₂ :: [ ]))  .
```

The following function compiles a polynomial, runs the program, and retrieves the resulting value from the heap:

```
compileRun : ∀ {n} → Vec Addr n → Addr → Polyⁿ Word → Heap → Word
compileRun rs r₀ e h =
  let ((r , ins) , _) = runSSA r₀ (compile _ rs e)
  in  runIns h ins !! r  .
```

Correctness Given a polynomial e, by correctness we intuitively mean that the compiled program computes the value which e would be evaluated to. A formal statement of correctness is complicated by the fact that $\mathsf{e} : \mathsf{Poly}^{\mathsf{n}}\ \mathsf{A}$ expects, as arguments, n polynomials arranged as a descending chain, each of them expects arguments as well, and ins expects their values to be stored in the heap.

Given a heap h, a chain es : DChain Word n, and a vector of addresses rs, the predicate Consistent h es rs holds if the values of each polynomial in es is stored in h at the corresponding address in rs. The predicate can be expressed by the following Agda datatype:

```
data Consistent (h : Heap) :
       ∀ {n} → DChain Word n → Vec Addr n → Set where
   []   : Consistent h tt []
   (::) : ∀ {n : ℕ} {es rs e r}
         → (h !! r ≡ sem n ringWord e es)
         → Consistent h      es        rs
         → Consistent h (e , es) (r :: rs)  .
```

Observe that in the definition of (::) the descending chain es is supplied to each invocation of sem to compute value of e, before e itself is accumulated to es.

The correctness of compile can be stated as:

```
compSem : ∀ (n : ℕ) {h : Heap}
   → (e : Polyⁿ Word)
   → (es : DChain Word n)
   → (rs : Vec Addr n) → (r₀ : Addr)
   → Consistent h es rs
   → NoOverlap r₀ rs
   → compileRun rs r₀ e h ≡ sem n e es  .
```

The predicate Consistent h es rs states that the values of the descending chain es are stored in the corresponding addresses rs. The predicate NoOverlap r_0 rs states that, if an SSA monad is run with starting address r_0, all subsequent allocated addresses will not overlap with those in rs. With the naive counter-based implementation of SSA, NoOverlap r_0 rs holds if r_0 is larger than every element in rs. The last line states that the polynomial e is compiled with argument addresses es and starting address r_0, and the value the program computes should be the same as the semantics of e, given the descending chain es as arguments.

With all the setting up, the property compSem n e can be proved by induction on n and e.

5 Conclusions and Related Work

In dependently typed programming, a typical choice in implementing multivariate polynomials is to represent de Bruin indices using Fin n, the type having exactly n members. This is done in, for example, the RingSolver in the Agda standard library [5], among many. The tagless-final representation [2] is another alternative. In this paper, we have explored yet another alternative, chosen to see how far we can go in exploiting the isomorphism between $R[X_1, X_2 \ldots, X_m]$ and univariate polynomial ring $R[X_1, X_2, \ldots, X_{m-1}][X_m]$. It turns out that we can go quite far — we managed to represent multivariate polynomials using univariate polynomials. Various operations on them can be defined inductively. In particular, we defined how a polynomial of vectors can be expanded to a vector of polynomials, and how a polynomial can be compiled to sequences of scalar-manipulating instructions like assembly-language programs. The correctness proofs of those operations also turn out to be straightforward inductions, once we figure out how to precisely express the correctness property.

We note that the current expansion formula is provided by the programmer. For example, in order to expand a complex polynomial expression into two real ones, the programmer needs to provide (in a RingVec) the formula $(a_1+b_1 i)(a_2+b_2 i) \bmod i^2+1 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1) i$. We can see that the divisor polynomial of the modular relationship can actually give rise to an equational type in which $i^2+1=0$, or any pair of polynomials are considered "equal" if their difference is a multiple of the polynomial i^2+1. In the future, we would like to further automate the derivation of this formula, so the programmer will only need to give us the definition of the equational types under consideration. The RingSolver [5] manipulates equations into normal forms to solve them, and the solution can be used in Agda programs by reflection. It is interesting to explore whether a similar approach may work for our purpose.

Acknowledgements The authors would like to thank the members of IFIP Working Group 2.1 for their valuable comments on the first presentation of this work.

References

1. R. S. Bird and O. de Moor. *Algebra of programming.* Prentice Hall International series in computer science. Prentice Hall, 1997.
2. J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.
3. A. I. Chen, C. O. Chen, M. Chen, C. Cheng, and B. Yang. Practical-sized instances of multivariate PKCs: Rainbow, TTS, and ℓIC-derivatives. In *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, pages 95–108. Springer, 2008.
4. E. Crockett and C. Peikert. $\Lambda o \lambda$: Functional Lattice Cryptography. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 993–1005. ACM, 2016.
5. N. A. Danielsson. Ring Solver, the Agda standard library. `https://github.com/agda/agda-stdlib/blob/master/src/Algebra/RingSolver.agda`.
6. T. Hungerford. *Algebra.* Graduate Texts in Mathematics. Springer New York, 2003.
7. V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors Over Rings. *IACR Cryptology ePrint Archive*, 2012:230, 2012.

8. V. S. Miller. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pages 417–426. Springer, 1985.
9. U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.